W3School Sql 教程

wizardforcel

Published
with GitBook



目錄

W3School SQL教程

来源：

- SQL教程
- MySQL教程
- SQLite教程

整理：飞龙

SQL基础

SQL 简介

SQL 是用于访问和处理数据库的标准的计算机语言。

什么是 **SQL**？

- SQL 指结构化查询语言
- SQL 使我们有能力访问数据库
- SQL 是一种 ANSI 的标准计算机语言

编者注：ANSI，美国国家标准化组织

SQL 能做什么？

- SQL 面向数据库执行查询
- SQL 可从数据库取回数据
- SQL 可在数据库中插入新的记录
- SQL 可更新数据库中的数据
- SQL 可从数据库删除记录
- SQL 可创建新数据库
- SQL 可在数据库中创建新表
- SQL 可在数据库中创建存储过程
- SQL 可在数据库中创建视图
- SQL 可以设置表、存储过程和视图的权限

SQL 是一种标准 - 但是...

SQL 是一门 ANSI 的标准计算机语言，用来访问和操作数据库系统。SQL 语句用于取回和更新数据库中的数据。SQL 可与数据库程序协同工作，比如 MS Access、DB2、Informix、MS SQL Server、Oracle、Sybase 以及其他数据库系统。

不幸地是，存在着很多不同版本的 SQL 语言，但是为了与 ANSI 标准相兼容，它们必须以相似的方式共同地来支持一些主要的关键词（比如 SELECT、UPDATE、DELETE、INSERT、WHERE 等等）。

注释：除了 SQL 标准之外，大部分 SQL 数据库程序都拥有它们自己的私有扩展！

在您的网站中使用 **SQL**

要创建发布数据库中数据的网站，您需要以下要素：

- RDBMS 数据库程序（比如 MS Access, SQL Server, MySQL）

- 服务器端脚本语言（比如 PHP 或 ASP）
- SQL
- HTML / CSS

RDBMS

RDBMS 指的是关系型数据库管理系统。

RDBMS 是 SQL 的基础，同样也是所有现代数据库系统的基础，比如 MS SQL Server, IBM DB2, Oracle, MySQL 以及 Microsoft Access。

RDBMS 中的数据存储在被称为表（tables）的数据库对象中。

表是相关的数据项的集合，它由列和行组成。

SQL 语法

数据库表

一个数据库通常包含一个或多个表。每个表由一个名字标识（例如“客户”或者“订单”）。表包含带有数据的记录（行）。

下面的例子是一个名为 "Persons" 的表：

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

上面的表包含三条记录（每一条对应一个人）和五个列（Id、姓、名、地址和城市）。

SQL 语句

您需要在数据库上执行的大部分工作都由 SQL 语句完成。

下面的语句从表中选取 LastName 列的数据：

```
SELECT LastName FROM Persons
```

结果集类似这样：

LastName
Adams
Bush
Carter

在本教程中，我们将为您讲解各种不同的 SQL 语句。

重要事项

一定要记住，*SQL* 对大小写不敏感！

SQL 语句后面的分号？

某些数据库系统要求在每条 SQL 命令的末端使用分号。在我们的教程中不使用分号。

分号是在数据库系统中分隔每条 SQL 语句的标准方法，这样就可以在对服务器的相同请求中执行一条以上的语句。

如果您使用的是 MS Access 和 SQL Server 2000，则不必在每条 SQL 语句之后使用分号，不过某些数据库软件要求必须使用分号。

SQL DML 和 DDL

可以把 SQL 分为两个部分：数据操作语言 (DML) 和 数据定义语言 (DDL)。

SQL (结构化查询语言)是用于执行查询的语法。但是 SQL 语言也包含用于更新、插入和删除记录的语法。

查询和更新指令构成了 SQL 的 DML 部分：

- *SELECT* - 从数据库表中获取数据
- *UPDATE* - 更新数据库表中的数据
- *DELETE* - 从数据库表中删除数据
- *INSERT INTO* - 向数据库表中插入数据

SQL 的数据定义语言 (DDL) 部分使我们有能力创建或删除表格。我们也可以定义索引（键），规定表之间的链接，以及施加表间的约束。

SQL 中最重要的 DDL 语句:

- *CREATE DATABASE* - 创建新数据库
- *ALTER DATABASE* - 修改数据库
- *CREATE TABLE* - 创建新表
- *ALTER TABLE* - 变更（改变）数据库表
- *DROP TABLE* - 删除表
- *CREATE INDEX* - 创建索引（搜索键）
- *DROP INDEX* - 删除索引

SQL SELECT 语句

本章讲解 **SELECT** 和 **SELECT *** 语句。

SQL SELECT 语句

SELECT 语句用于从表中选取数据。

结果被存储在一个结果表中（称为结果集）。

SQL SELECT 语法

```
SELECT 列名称 FROM 表名称
```

以及：

```
SELECT * FROM 表名称
```

注释：SQL 语句对大小写不敏感。SELECT 等效于 select。

SQL SELECT 实例

如需获取名为 "LastName" 和 "FirstName" 的列的内容（从名为 "Persons" 的数据库表），请使用类似这样的 SELECT 语句：

```
SELECT LastName,FirstName FROM Persons
```

"Persons" 表:

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

结果：

LastName	FirstName
Adams	John
Bush	George
Carter	Thomas

SQL SELECT * 实例

现在我们希望从 "Persons" 表中选取所有的列。

请使用符号 * 取代列的名称，就像这样：

```
SELECT * FROM Persons
```

提示：星号（*）是选取所有列的快捷方式。

结果：

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

在结果集（result-set）中导航

由 SQL 查询程序获得的结果被存放在一个结果集中。大多数数据库软件系统都允许使用编程函数在结果集中进行导航，比如：Move-To-First-Record、Get-Record-Content、Move-To-Next-Record 等等。

类似这些编程函数不在本教程讲解之列。如需学习通过函数调用访问数据的知识，请访问我们的 ADO 教程 和 PHP 教程。

SQL SELECT DISTINCT 语句

本章讲解 **SELECT DISTINCT** 语句。

SQL SELECT DISTINCT 语句

在表中，可能会包含重复值。这并不成问题，不过，有时您也许希望仅仅列出不同（distinct）的值。

关键词 DISTINCT 用于返回唯一不同的值。

语法：

```
SELECT DISTINCT 列名称 FROM 表名称
```

使用 DISTINCT 关键词

如果要从 "Company" 列中选取所有的值，我们需要使用 SELECT 语句：

```
SELECT Company FROM Orders
```

"Orders"表：

Company	OrderNumber
IBM	3532
W3School	2356
Apple	4698
W3School	6953

结果：

Company
IBM
W3School
Apple
W3School

请注意，在结果集中，W3School 被列出了两次。

如需从 Company" 列中仅选取唯一不同的值，我们需要使用 SELECT DISTINCT 语句：

```
SELECT DISTINCT Company FROM Orders
```

结果：

Company
IBM
W3School
Apple

现在，在结果集中，"W3School" 仅被列出了一次。

SQL WHERE 子句

WHERE 子句用于规定选择的标准。

WHERE 子句

如需有条件地从表中选取数据，可将 WHERE 子句添加到 SELECT 语句。

语法

```
SELECT 列名称 FROM 表名称 WHERE 列 运算符 值
```

下面的运算符可在 WHERE 子句中使用：

操作符	描述
=	等于
<>	不等于
>	大于
<	小于
>=	大于等于
<=	小于等于
BETWEEN	在某个范围内
LIKE	搜索某种模式

注释：在某些版本的 SQL 中，操作符 <> 可以写为 !=。

使用 WHERE 子句

如果只希望选取居住在城市 "Beijing" 中的人，我们需要向 SELECT 语句添加 WHERE 子句：

```
SELECT * FROM Persons WHERE City='Beijing'
```

"Persons" 表

LastName	FirstName	Address	City	Year
Adams	John	Oxford Street	London	1970
Bush	George	Fifth Avenue	New York	1975
Carter	Thomas	Changan Street	Beijing	1980
Gates	Bill	Xuanwumen 10	Beijing	1985

结果：

LastName	FirstName	Address	City	Year
Carter	Thomas	Changan Street	Beijing	1980
Gates	Bill	Xuanwumen 10	Beijing	1985

引号的使用

请注意，我们在例子中的条件值周围使用的是单引号。

SQL 使用单引号来环绕文本值（大部分数据库系统也接受双引号）。如果是数值，请不要使用引号。

文本值：

```
这是正确的：
SELECT * FROM Persons WHERE FirstName='Bush'

这是错误的：
SELECT * FROM Persons WHERE FirstName=Bush
```

数值：

```
这是正确的：
SELECT * FROM Persons WHERE Year>1965

这是错误的：
SELECT * FROM Persons WHERE Year>'1965'
```

SQL AND & OR 运算符

AND 和 OR 运算符用于基于一个以上的条件对记录进行过滤。

AND 和 OR 运算符

AND 和 OR 可在 WHERE 子语句中把两个或多个条件结合起来。

如果第一个条件和第二个条件都成立，则 AND 运算符显示一条记录。

如果第一个条件和第二个条件中只要有一个成立，则 OR 运算符显示一条记录。

原始的表 (用在例子中的)：

LastName	FirstName	Address	City
Adams	John	Oxford Street	London
Bush	George	Fifth Avenue	New York
Carter	Thomas	Changan Street	Beijing
Carter	William	Xuanwumen 10	Beijing

AND 运算符实例

使用 AND 来显示所有姓为 "Carter" 并且名为 "Thomas" 的人：

```
SELECT * FROM Persons WHERE FirstName='Thomas' AND LastName='Carte
```

结果：

LastName	FirstName	Address	City
Carter	Thomas	Changan Street	Beijing

OR 运算符实例

使用 OR 来显示所有姓为 "Carter" 或者名为 "Thomas" 的人：

```
SELECT * FROM Persons WHERE firstname='Thomas' OR lastname='Carter
```

结果：

LastName	FirstName	Address	City
Carter	Thomas	Changan Street	Beijing
Carter	William	Xuanwumen 10	Beijing

结合 AND 和 OR 运算符

我们也可以把 AND 和 OR 结合起来（使用圆括号来组成复杂的表达式）：

```
SELECT * FROM Persons WHERE (FirstName='Thomas' OR FirstName='Willi
AND LastName='Carter'
```

结果：

LastName	FirstName	Address	City
Carter	Thomas	Changan Street	Beijing
Carter	William	Xuanwumen 10	Beijing

SQL ORDER BY 子句

ORDER BY 语句用于对结果集进行排序。

ORDER BY 语句

ORDER BY 语句用于根据指定的列对结果集进行排序。

ORDER BY 语句默认按照升序对记录进行排序。

如果您希望按照降序对记录进行排序，可以使用 DESC 关键字。

原始的表 (用在例子中的)：

Orders 表:

Company	OrderNumber
IBM	3532
W3School	2356
Apple	4698
W3School	6953

实例 1

以字母顺序显示公司名称：

```
SELECT Company, OrderNumber FROM Orders ORDER BY Company
```

结果：

Company	OrderNumber
Apple	4698
IBM	3532
W3School	6953
W3School	2356

实例 2

以字母顺序显示公司名称（Company），并以数字顺序显示顺序号（OrderNumber）：

```
SELECT Company, OrderNumber FROM Orders ORDER BY Company, OrderNumb
```

结果：

Company	OrderNumber
Apple	4698
IBM	3532
W3School	2356
W3School	6953

实例 3

以逆字母顺序显示公司名称：

```
SELECT Company, OrderNumber FROM Orders ORDER BY Company DESC
```

结果：

Company	OrderNumber
W3School	6953
W3School	2356
IBM	3532
Apple	4698

实例 4

以逆字母顺序显示公司名称，并以数字顺序显示顺序号：

```
SELECT Company, OrderNumber FROM Orders ORDER BY Company DESC, Orde
```

结果：

Company	OrderNumber
W3School	2356
W3School	6953
IBM	3532
Apple	4698

注意：在以上的结果中有两个相等的公司名称 (W3School)。只有这一次，在第一列中有相同的值时，第二列是以升序排列的。如果第一列中有些值为 nulls 时，情况也是这样的。

SQL INSERT INTO 语句

INSERT INTO 语句

INSERT INTO 语句用于向表格中插入新的行。

语法

```
INSERT INTO 表名称 VALUES (值1, 值2,....)
```

我们也可以指定所要插入数据的列：

```
INSERT INTO table_name (列1, 列2,...) VALUES (值1, 值2,....)
```

插入新的行

"Persons" 表：

LastName	FirstName	Address	City
Carter	Thomas	Changan Street	Beijing

SQL 语句：

```
INSERT INTO Persons VALUES ('Gates', 'Bill', 'Xuanwumen 10', 'Beiji
```

结果：

LastName	FirstName	Address	City
Carter	Thomas	Changan Street	Beijing
Gates	Bill	Xuanwumen 10	Beijing

在指定的列中插入数据

"Persons" 表：

LastName	FirstName	Address	City
Carter	Thomas	Changan Street	Beijing
Gates	Bill	Xuanwumen 10	Beijing

SQL 语句：

```
INSERT INTO Persons (LastName, Address) VALUES ('Wilson', 'Champs-E
```

结果：

LastName	FirstName	Address	City
Carter	Thomas	Changan Street	Beijing
Gates	Bill	Xuanwumen 10	Beijing
Wilson	Champs-Elysees		

SQL UPDATE 语句

Update 语句

Update 语句用于修改表中的数据。

语法：

```
UPDATE 表名称 SET 列名称 = 新值 WHERE 列名称 = 某值
```

Person:

LastName	FirstName	Address	City
Gates	Bill	Xuanwumen 10	Beijing
Wilson	Champs-Elysees		

更新某一行中的一个列

我们为 lastname 是 "Wilson" 的人添加 firstname：

```
UPDATE Person SET FirstName = 'Fred' WHERE LastName = 'Wilson'
```

结果：

LastName	FirstName	Address	City
Gates	Bill	Xuanwumen 10	Beijing
Wilson	Fred	Champs-Elysees	

更新某一行中的若干列

我们会修改地址（address），并添加城市名称（city）：

```
UPDATE Person SET Address = 'Zhongshan 23', City = 'Nanjing'
WHERE LastName = 'Wilson'
```

结果：

LastName	FirstName	Address	City
Gates	Bill	Xuanwumen 10	Beijing
Wilson	Fred	Zhongshan 23	Nanjing

SQL DELETE 语句

DELETE 语句

DELETE 语句用于删除表中的行。

语法

```
DELETE FROM 表名称 WHERE 列名称 = 值
```

Person:

LastName	FirstName	Address	City
Gates	Bill	Xuanwumen 10	Beijing
Wilson	Fred	Zhongshan 23	Nanjing

删除某行

"Fred Wilson" 会被删除：

```
DELETE FROM Person WHERE LastName = 'Wilson'
```

结果:

LastName	FirstName	Address	City
Gates	Bill	Xuanwumen 10	Beijing

删除所有行

可以在不删除表的情况下删除所有的行。这意味着表的结构、属性和索引都是完整的：

```
DELETE FROM table_name
```

或者：

```
DELETE * FROM table_name
```

SQL高级

SQL TOP 子句

TOP 子句

TOP 子句用于规定要返回的记录的数目。

对于拥有数千条记录的大型表来说，TOP 子句是非常有用的。

注释：并非所有的数据库系统都支持 TOP 子句。

SQL Server 的语法：

```
SELECT TOP number|percent column_name(s)
FROM table_name
```

MySQL 和 Oracle 中的 SQL SELECT TOP 是等价的

MySQL 语法

```
SELECT column_name(s)
FROM table_name
LIMIT number
```

例子

```
SELECT *
FROM Persons
LIMIT 5
```

Oracle 语法

```
SELECT column_name(s)
FROM table_name
WHERE ROWNUM <= number
```

例子

```
SELECT *
FROM Persons
WHERE ROWNUM <= 5
```

原始的表 (用在例子中的)：

Persons 表:

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing
4	Obama	Barack	Pennsylvania Avenue	Washington

SQL TOP 实例

现在，我们希望从上面的 "Persons" 表中选取头两条记录。

我们可以使用下面的 SELECT 语句：

```
SELECT TOP 2 * FROM Persons
```

结果：

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York

SQL TOP PERCENT 实例

现在，我们希望从上面的 "Persons" 表中选取 50% 的记录。

我们可以使用下面的 SELECT 语句：

```
SELECT TOP 50 PERCENT * FROM Persons
```

结果：

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York

SQL LIKE 操作符

LIKE 操作符用于在 **WHERE** 子句中搜索列中的指定模式。

LIKE 操作符

LIKE 操作符用于在 WHERE 子句中搜索列中的指定模式。

SQL LIKE 操作符语法

```
SELECT column_name(s)
FROM table_name
WHERE column_name LIKE pattern
```

原始的表 (用在例子中的)：

Persons 表:

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

LIKE 操作符实例

例子 1

现在，我们希望从上面的 "Persons" 表中选取居住在以 "N" 开始的城市里的人：

我们可以使用下面的 SELECT 语句：

```
SELECT * FROM Persons
WHERE City LIKE 'N%'
```

提示："%" 可用于定义通配符（模式中缺少的字母）。

结果集：

Id	LastName	FirstName	Address	City
2	Bush	George	Fifth Avenue	New York

例子 2

接下来，我们希望从 "Persons" 表中选取居住在以 "g" 结尾的城市里的人：

我们可以使用下面的 SELECT 语句：

```
SELECT * FROM Persons
WHERE City LIKE '%g'
```

结果集：

Id	LastName	FirstName	Address	City
3	Carter	Thomas	Changan Street	Beijing

例子 3

接下来，我们希望从 "Persons" 表中选取居住在包含 "lon" 的城市里的人：

我们可以使用下面的 SELECT 语句：

```
SELECT * FROM Persons
WHERE City LIKE '%lon%'
```

结果集：

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London

例子 4

通过使用 NOT 关键字，我们可以从 "Persons" 表中选取居住在不包含 "lon" 的城市里的人：

我们可以使用下面的 SELECT 语句：

```
SELECT * FROM Persons
WHERE City NOT LIKE '%lon%'
```

结果集：

Id	LastName	FirstName	Address	City
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

SQL 通配符

在搜索数据库中的数据时，您可以使用 **SQL** 通配符。

SQL 通配符

在搜索数据库中的数据时，SQL 通配符可以替代一个或多个字符。

SQL 通配符必须与 LIKE 运算符一起使用。

在 SQL 中，可使用以下通配符：

通配符	描述
%	替代一个或多个字符
_	仅替代一个字符
[charlist]	字符列中的任何单一字符
[^charlist] 或者 [!charlist]	不在字符列中的任何单一字符

原始的表 (用在例子中的)： **Persons 表:**

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

使用 % 通配符

例子 1

现在，我们希望从上面的 "Persons" 表中选取居住在以 "Ne" 开始的城市里的人：

我们可以使用下面的 SELECT 语句：

```
SELECT * FROM Persons
WHERE City LIKE 'Ne%'
```

结果集：

Id	LastName	FirstName	Address	City
2	Bush	George	Fifth Avenue	New York

例子 2

接下来，我们希望从 "Persons" 表中选取居住在包含 "lond" 的城市里的人：

我们可以使用下面的 SELECT 语句：

```
SELECT * FROM Persons
WHERE City LIKE '%lond%'
```

结果集：

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London

使用 _ 通配符

例子 1

现在，我们希望从上面的 "Persons" 表中选取名字的第一个字符之后是 "eorge" 的人：

我们可以使用下面的 SELECT 语句：

```
SELECT * FROM Persons
WHERE FirstName LIKE '_eorge'
```

结果集：

Id	LastName	FirstName	Address	City
2	Bush	George	Fifth Avenue	New York

例子 2

接下来，我们希望从 "Persons" 表中选取的这条记录的姓氏以 "C" 开头，然后是一个任意字符，然后是 "r"，然后是任意字符，然后是 "er"：

我们可以使用下面的 SELECT 语句：

```
SELECT * FROM Persons
WHERE LastName LIKE 'C_r_er'
```

结果集：

Id	LastName	FirstName	Address	City
3	Carter	Thomas	Changan Street	Beijing

使用 [charlist] 通配符

例子 1

现在，我们希望从上面的 "Persons" 表中选取居住的城市以 "A" 或 "L" 或 "N" 开头的人：

我们可以使用下面的 SELECT 语句：

```
SELECT * FROM Persons
WHERE City LIKE '[ALN]%'
```

结果集：

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York

例子 2

现在，我们希望从上面的 "Persons" 表中选取居住的城市不以 "A" 或 "L" 或 "N" 开头的人：

我们可以使用下面的 SELECT 语句：

```
SELECT * FROM Persons
WHERE City LIKE '[!ALN]%'
```

结果集：

Id	LastName	FirstName	Address	City
3	Carter	Thomas	Changan Street	Beijing

SQL IN 操作符

IN 操作符

IN 操作符允许我们在 WHERE 子句中规定多个值。

SQL IN 语法

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (value1,value2,...)
```

原始的表 (在实例中使用：)

Persons 表:

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

IN 操作符实例

现在，我们希望从上表中选取姓氏为 Adams 和 Carter 的人：

我们可以使用下面的 SELECT 语句：

```
SELECT * FROM Persons
WHERE LastName IN ('Adams','Carter')
```

结果集：

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
3	Carter	Thomas	Changan Street	Beijing

SQL BETWEEN 操作符

BETWEEN 操作符在 **WHERE** 子句中使用，作用是选取介于两个值之间的数据范围。

BETWEEN 操作符

操作符 BETWEEN ... AND 会选取介于两个值之间的数据范围。这些值可以是数值、文本或者日期。

SQL BETWEEN 语法

```
SELECT column_name(s)
FROM table_name
WHERE column_name
BETWEEN value1 AND value2
```

原始的表 (在实例中使用：)

Persons 表:

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing
4	Gates	Bill	Xuanwumen 10	Beijing

BETWEEN 操作符实例

如需以字母顺序显示介于 "Adams"（包括）和 "Carter"（不包括）之间的人，请使用下面的 SQL：

```
SELECT * FROM Persons
WHERE LastName
BETWEEN 'Adams' AND 'Carter'
```

结果集：

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York

重要事项：不同的数据库对 BETWEEN...AND 操作符的处理方式是有差异的。某些数据库会列出介于 "Adams" 和 "Carter" 之间的人，但不包括 "Adams" 和 "Carter" ；某些数据库会列出介于 "Adams" 和 "Carter" 之间并包括 "Adams" 和 "Carter" 的人；而另一些数据库会列出介于 "Adams" 和 "Carter" 之间的人，包括 "Adams" ，但不包括 "Carter" 。

所以，请检查你的数据库是如何处理 BETWEEN....AND 操作符的！

实例 2

如需使用上面的例子显示范围之外的人，请使用 NOT 操作符：

```
SELECT * FROM Persons
WHERE LastName
NOT BETWEEN 'Adams' AND 'Carter'
```

结果集：

Id	LastName	FirstName	Address	City
3	Carter	Thomas	Changan Street	Beijing
4	Gates	Bill	Xuanwumen 10	Beijing

SQL Alias（别名）

通过使用 **SQL**，可以为列名称和表名称指定别名（**Alias**）。

SQL Alias

表的 SQL Alias 语法

```
SELECT column_name(s)
FROM table_name
AS alias_name
```

列的 SQL Alias 语法

```
SELECT column_name AS alias_name
FROM table_name
```

Alias 实例: 使用表名称别名

假设我们有两个表分别是："Persons" 和 "Product_Orders"。我们分别为它们指定别名 "p" 和 "po"。

现在，我们希望列出 "John Adams" 的所有定单。

我们可以使用下面的 SELECT 语句：

```
SELECT po.OrderID, p.LastName, p.FirstName
FROM Persons AS p, Product_Orders AS po
WHERE p.LastName='Adams' AND p.FirstName='John'
```

不使用别名的 SELECT 语句：

```
SELECT Product_Orders.OrderID, Persons.LastName, Persons.FirstName
FROM Persons, Product_Orders
WHERE Persons.LastName='Adams' AND Persons.FirstName='John'
```

从上面两条 SELECT 语句您可以看到，别名使查询程序更易阅读和书写。

Alias 实例: 使用一个列名别名

表 Persons:

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

SQL:

```
SELECT LastName AS Family, FirstName AS Name
FROM Persons
```

结果：

Family	Name
Adams	John
Bush	George
Carter	Thomas

SQL JOIN

SQL join 用于根据两个或多个表中的列之间的关系，从这些表中查询数据。

Join 和 Key

有时为了得到完整的结果，我们需要从两个或更多的表中获取结果。我们就需要执行 join。

数据库中的表可通过键将彼此联系起来。主键（Primary Key）是一个列，在这个列中的每一行的值都是唯一的。在表中，每个主键的值都是唯一的。这样做的目的是在不重复每个表中的所有数据的情况下，把表间的数据交叉捆绑在一起。

请看 "Persons" 表：

Id_P	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

请注意，"Id_P" 列是 Persons 表中的的主键。这意味着没有两行能够拥有相同的 Id_P。即使两个人的姓名完全相同，Id_P 也可以区分他们。

接下来请看 "Orders" 表：

Id_O	OrderNo	Id_P
1	77895	3
2	44678	3
3	22456	1
4	24562	1
5	34764	65

请注意，"Id_O" 列是 Orders 表中的的主键，同时，"Orders" 表中的 "Id_P" 列用于引用 "Persons" 表中的人，而无需使用他们的确切姓名。

请留意，"Id_P" 列把上面的两个表联系了起来。

引用两个表

我们可以通过引用两个表的方式，从两个表中获取数据：

谁订购了产品，并且他们订购了什么产品？

```
SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo
FROM Persons, Orders
WHERE Persons.Id_P = Orders.Id_P
```

结果集：

LastName	FirstName	OrderNo
Adams	John	22456
Adams	John	24562
Carter	Thomas	77895
Carter	Thomas	44678

SQL JOIN - 使用 Join

除了上面的方法，我们也可以使用关键词 JOIN 来从两个表中获取数据。

如果我们希望列出所有人的定购，可以使用下面的 SELECT 语句：

```
SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo
FROM Persons
INNER JOIN Orders
ON Persons.Id_P = Orders.Id_P
ORDER BY Persons.LastName
```

结果集：

LastName	FirstName	OrderNo
Adams	John	22456
Adams	John	24562
Carter	Thomas	77895
Carter	Thomas	44678

不同的 SQL JOIN

除了我们在上面的例子中使用的 INNER JOIN（内连接），我们还可以使用其他几种连接。

下面列出了您可以使用的 JOIN 类型，以及它们之间的差异。

- JOIN: 如果表中有至少一个匹配，则返回行
- LEFT JOIN: 即使右表中没有匹配，也从左表返回所有的行
- RIGHT JOIN: 即使左表中没有匹配，也从右表返回所有的行
- FULL JOIN: 只要其中一个表中存在匹配，就返回行

SQL INNER JOIN 关键字

SQL INNER JOIN 关键字

在表中存在至少一个匹配时，INNER JOIN 关键字返回行。

INNER JOIN 关键字语法

```
SELECT column_name(s)
FROM table_name1
INNER JOIN table_name2
ON table_name1.column_name=table_name2.column_name
```

注释：INNER JOIN 与 JOIN 是相同的。

原始的表 (用在例子中的)：

"Persons" 表：

Id_P	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

"Orders" 表：

Id_O	OrderNo	Id_P
1	77895	3
2	44678	3
3	22456	1
4	24562	1
5	34764	65

内连接（INNER JOIN）实例

现在，我们希望列出所有人的定购。

您可以使用下面的 SELECT 语句：

```
SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo
FROM Persons
INNER JOIN Orders
ON Persons.Id_P=Orders.Id_P
ORDER BY Persons.LastName
```

结果集：

LastName	FirstName	OrderNo
Adams	John	22456
Adams	John	24562
Carter	Thomas	77895
Carter	Thomas	44678

INNER JOIN 关键字在表中存在至少一个匹配时返回行。如果 "Persons" 中的行在 "Orders" 中没有匹配，就不会列出这些行。

SQL LEFT JOIN 关键字

SQL LEFT JOIN 关键字

LEFT JOIN 关键字会从左表 (table_name1) 那里返回所有的行，即使在右表 (table_name2) 中没有匹配的行。

LEFT JOIN 关键字语法

```
SELECT column_name(s)
FROM table_name1
LEFT JOIN table_name2
ON table_name1.column_name=table_name2.column_name
```

注释：在某些数据库中， LEFT JOIN 称为 LEFT OUTER JOIN。

原始的表 (用在例子中的)：

"Persons" 表：

Id_P	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

"Orders" 表：

Id_O	OrderNo	Id_P
1	77895	3
2	44678	3
3	22456	1
4	24562	1
5	34764	65

左连接（LEFT JOIN）实例

现在，我们希望列出所有的人，以及他们的定购 - 如果有的话。

您可以使用下面的 SELECT 语句：

```
SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo
FROM Persons
LEFT JOIN Orders
ON Persons.Id_P=Orders.Id_P
ORDER BY Persons.LastName
```

结果集：

LastName	FirstName	OrderNo
Adams	John	22456
Adams	John	24562
Carter	Thomas	77895
Carter	Thomas	44678
Bush	George	

LEFT JOIN 关键字会从左表 (Persons) 那里返回所有的行，即使在右表 (Orders) 中没有匹配的行。

SQL RIGHT JOIN 关键字

SQL RIGHT JOIN 关键字

RIGHT JOIN 关键字会右表 (table_name2) 那里返回所有的行，即使在左表 (table_name1) 中没有匹配的行。

RIGHT JOIN 关键字语法

```
SELECT column_name(s)
FROM table_name1
RIGHT JOIN table_name2
ON table_name1.column_name=table_name2.column_name
```

注释：在某些数据库中， RIGHT JOIN 称为 RIGHT OUTER JOIN。

原始的表 (用在例子中的)：

"Persons" 表：

Id_P	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

"Orders" 表：

Id_O	OrderNo	Id_P
1	77895	3
2	44678	3
3	22456	1
4	24562	1
5	34764	65

右连接（RIGHT JOIN）实例

现在，我们希望列出所有的定单，以及定购它们的人 - 如果有的话。

您可以使用下面的 SELECT 语句：

```
SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo
FROM Persons
RIGHT JOIN Orders
ON Persons.Id_P=Orders.Id_P
ORDER BY Persons.LastName
```

结果集：

LastName	FirstName	OrderNo
Adams	John	22456
Adams	John	24562
Carter	Thomas	77895
Carter	Thomas	44678
34764		

RIGHT JOIN 关键字会从右表 (Orders) 那里返回所有的行，即使在左表 (Persons) 中没有匹配的行。

SQL FULL JOIN 关键字

SQL FULL JOIN 关键字

只要其中某个表存在匹配，FULL JOIN 关键字就会返回行。

FULL JOIN 关键字语法

```
SELECT column_name(s)
FROM table_name1
FULL JOIN table_name2
ON table_name1.column_name=table_name2.column_name
```

注释：在某些数据库中， FULL JOIN 称为 FULL OUTER JOIN。

原始的表 (用在例子中的)：

"Persons" 表：

Id_P	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

"Orders" 表：

Id_O	OrderNo	Id_P
1	77895	3
2	44678	3
3	22456	1
4	24562	1
5	34764	65

全连接（FULL JOIN）实例

现在，我们希望列出所有的人，以及他们的定单，以及所有的定单，以及定购它们的人。

您可以使用下面的 SELECT 语句：

```
SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo
FROM Persons
FULL JOIN Orders
ON Persons.Id_P=Orders.Id_P
ORDER BY Persons.LastName
```

结果集：

LastName	FirstName	OrderNo
Adams	John	22456
Adams	John	24562
Carter	Thomas	77895
Carter	Thomas	44678
Bush	George	
34764		

FULL JOIN 关键字会从左表 (Persons) 和右表 (Orders) 那里返回所有的行。如果 "Persons" 中的行在表 "Orders" 中没有匹配，或者如果 "Orders" 中的行在表 "Persons" 中没有匹配，这些行同样会列出。

SQL UNION 和 UNION ALL 操作符

SQL UNION 操作符

UNION 操作符用于合并两个或多个 SELECT 语句的结果集。

请注意，UNION 内部的 SELECT 语句必须拥有相同数量的列。列也必须拥有相似的数据类型。同时，每条 SELECT 语句中的列的顺序必须相同。

SQL UNION 语法

```
SELECT column_name(s) FROM table_name1
UNION
SELECT column_name(s) FROM table_name2
```

注释：默认地，UNION 操作符选取不同的值。如果允许重复的值，请使用 UNION ALL。

SQL UNION ALL 语法

```
SELECT column_name(s) FROM table_name1
UNION ALL
SELECT column_name(s) FROM table_name2
```

另外，UNION 结果集中的列名总是等于 UNION 中第一个 SELECT 语句中的列名。

下面的例子中使用的原始表：

Employees_China:

E_ID	E_Name
01	Zhang, Hua
02	Wang, Wei
03	Carter, Thomas
04	Yang, Ming

Employees_USA:

E_ID	E_Name
01	Adams, John
02	Bush, George
03	Carter, Thomas
04	Gates, Bill

使用 UNION 命令

实例

列出所有在中国和美国的不同的雇员名：

```
SELECT E_Name FROM Employees_China
UNION
SELECT E_Name FROM Employees_USA
```

结果

E_Name
Zhang, Hua
Wang, Wei
Carter, Thomas
Yang, Ming
Adams, John
Bush, George
Gates, Bill

注释：这个命令无法列出在中国和美国的所有雇员。在上面的例子中，我们有两个名字相同的雇员，他们当中只有一个人被列出来了。UNION 命令只会选取不同的值。

UNION ALL

UNION ALL 命令和 UNION 命令几乎是等效的，不过 UNION ALL 命令会列出所有的值。

```
SQL Statement 1
UNION ALL
SQL Statement 2
```

使用 UNION ALL 命令

实例：

列出在中国和美国的所有的雇员：

```
SELECT E_Name FROM Employees_China
UNION ALL
SELECT E_Name FROM Employees_USA
```

结果

E_Name
Zhang, Hua
Wang, Wei
Carter, Thomas
Yang, Ming
Adams, John
Bush, George
Carter, Thomas
Gates, Bill

SQL SELECT INTO 语句

SQL SELECT INTO 语句可用于创建表的备份复件。

SELECT INTO 语句

SELECT INTO 语句从一个表中选取数据，然后把数据插入另一个表中。

SELECT INTO 语句常用于创建表的备份复件或者用于对记录进行存档。

SQL SELECT INTO 语法

您可以把所有的列插入新表：

```
SELECT *
INTO new_table_name [IN externaldatabase]
FROM old_tablename
```

或者只把希望的列插入新表：

```
SELECT column_name(s)
INTO new_table_name [IN externaldatabase]
FROM old_tablename
```

SQL SELECT INTO 实例 - 制作备份复件

下面的例子会制作 "Persons" 表的备份复件：

```
SELECT *
INTO Persons_backup
FROM Persons
```

IN 子句可用于向另一个数据库中拷贝表：

```
SELECT *
INTO Persons IN 'Backup.mdb'
FROM Persons
```

如果我们希望拷贝某些域，可以在 SELECT 语句后列出这些域：

```
SELECT LastName,FirstName
INTO Persons_backup
FROM Persons
```

SQL SELECT INTO 实例 - 带有 WHERE 子句

我们也可以添加 WHERE 子句。

下面的例子通过从 "Persons" 表中提取居住在 "Beijing" 的人的信息，创建了一个带有两个列的名为 "Persons_backup" 的表：

```
SELECT LastName,Firstname
INTO Persons_backup
FROM Persons
WHERE City='Beijing'
```

SQL SELECT INTO 实例 - 被连接的表

从一个以上的表中选取数据也是可以做到的。

下面的例子会创建一个名为 "Persons_Order_Backup" 的新表，其中包含了从 Persons 和 Orders 两个表中取得的信息：

```
SELECT Persons.LastName,Orders.OrderNo
INTO Persons_Order_Backup
FROM Persons
INNER JOIN Orders
ON Persons.Id_P=Orders.Id_P
```

SQL CREATE DATABASE 语句

CREATE DATABASE 语句

CREATE DATABASE 用于创建数据库。

SQL CREATE DATABASE 语法

```
CREATE DATABASE database_name
```

SQL CREATE DATABASE 实例

现在我们希望创建一个名为 "my_db" 的数据库。

我们使用下面的 CREATE DATABASE 语句：

```
CREATE DATABASE my_db
```

可以通过 CREATE TABLE 来添加数据库表。

SQL CREATE TABLE 语句

CREATE TABLE 语句

CREATE TABLE 语句用于创建数据库中的表。

SQL CREATE TABLE 语法

```
CREATE TABLE 表名称
(
列名称1 数据类型,
列名称2 数据类型,
列名称3 数据类型,
....
)
```

数据类型（data_type）规定了列可容纳何种数据类型。下面的表格包含了SQL中最常用的数据类型：

数据类型	描述
integer(size) int(size) smallint(size) tinyint(size)	仅容纳整数。在括号内规定数字的最大位数。
decimal(size,d) numeric(size,d)	容纳带有小数的数字。 "size" 规定数字的最大位数。"d" 规定小数点右侧的最大位数。
char(size)	容纳固定长度的字符串（可容纳字母、数字以及特殊字符）。 在括号中规定字符串的长度。
varchar(size)	容纳可变长度的字符串（可容纳字母、数字以及特殊的字符）。 在括号中规定字符串的最大长度。
date(yyyymmdd)	容纳日期。

SQL CREATE TABLE 实例

本例演示如何创建名为 "Person" 的表。

该表包含 5 个列，列名分别是："Id_P"、"LastName"、"FirstName"、"Address" 以及 "City"：

```
CREATE TABLE Persons
(
Id_P int,
LastName varchar(255),
FirstName varchar(255),
Address varchar(255),
City varchar(255)
)
```

Id_P 列的数据类型是 int，包含整数。其余 4 列的数据类型是 varchar，最大长度为 255 个字符。

空的 "Persons" 表类似这样：

Id_P	LastName	FirstName	Address	City

可使用 INSERT INTO 语句向空表写入数据。

SQL 约束 (Constraints)

SQL 约束

约束用于限制加入表的数据的类型。

可以在创建表时规定约束（通过 CREATE TABLE 语句），或者在表创建之后也可以（通过 ALTER TABLE 语句）。

我们将主要探讨以下几种约束：

- NOT NULL
- UNIQUE
- PRIMARY KEY
- FOREIGN KEY
- CHECK
- DEFAULT

注释：在下面的章节，我们会详细讲解每一种约束。

SQL NOT NULL 约束

SQL NOT NULL 约束

NOT NULL 约束强制列不接受 NULL 值。

NOT NULL 约束强制字段始终包含值。这意味着，如果不向字段添加值，就无法插入新记录或者更新记录。

下面的 SQL 语句强制 "Id_P" 列和 "LastName" 列不接受 NULL 值：

```
CREATE TABLE Persons
(
Id_P int NOT NULL,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255)
)
```

SQL UNIQUE 约束

SQL UNIQUE 约束

UNIQUE 约束唯一标识数据库表中的每条记录。

UNIQUE 和 PRIMARY KEY 约束均为列或列集合提供了唯一性的保证。

PRIMARY KEY 拥有自动定义的 UNIQUE 约束。

请注意，每个表可以有多个 UNIQUE 约束，但是每个表只能有一个 PRIMARY KEY 约束。

SQL UNIQUE Constraint on CREATE TABLE

下面的 SQL 在 "Persons" 表创建时在 "Id_P" 列创建 UNIQUE 约束：

MySQL:

```
CREATE TABLE Persons
(
Id_P int NOT NULL,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255),
UNIQUE (Id_P)
)
```

SQL Server / Oracle / MS Access:

```
CREATE TABLE Persons
(
Id_P int NOT NULL UNIQUE,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255)
)
```

如果需要命名 UNIQUE 约束，以及为多个列定义 UNIQUE 约束，请使用下面的 SQL 语法：

MySQL / SQL Server / Oracle / MS Access:

```
CREATE TABLE Persons
(
Id_P int NOT NULL,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255),
CONSTRAINT uc_PersonID UNIQUE (Id_P,LastName)
)
```

SQL UNIQUE Constraint on ALTER TABLE

当表已被创建时，如需在 "Id_P" 列创建 UNIQUE 约束，请使用下列 SQL：

MySQL / SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
ADD UNIQUE (Id_P)
```

如需命名 UNIQUE 约束，并定义多个列的 UNIQUE 约束，请使用下面的 SQL 语法：

MySQL / SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
ADD CONSTRAINT uc_PersonID UNIQUE (Id_P,LastName)
```

撤销 UNIQUE 约束

如需撤销 UNIQUE 约束，请使用下面的 SQL：

MySQL:

```
ALTER TABLE Persons
DROP INDEX uc_PersonID
```

SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
DROP CONSTRAINT uc_PersonID
```

SQL PRIMARY KEY 约束

SQL PRIMARY KEY 约束

PRIMARY KEY 约束唯一标识数据库表中的每条记录。

主键必须包含唯一的值。

主键列不能包含 NULL 值。

每个表都应该有一个主键，并且每个表只能有一个主键。

SQL PRIMARY KEY Constraint on CREATE TABLE

下面的 SQL 在 "Persons" 表创建时在 "Id_P" 列创建 PRIMARY KEY 约束：

MySQL:

```
CREATE TABLE Persons
(
Id_P int NOT NULL,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255),
PRIMARY KEY (Id_P)
)
```

SQL Server / Oracle / MS Access:

```
CREATE TABLE Persons
(
Id_P int NOT NULL PRIMARY KEY,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255)
)
```

如果需要命名 PRIMARY KEY 约束，以及为多个列定义 PRIMARY KEY 约束，请使用下面的 SQL 语法：

MySQL / SQL Server / Oracle / MS Access:

```
CREATE TABLE Persons
(
Id_P int NOT NULL,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255),
CONSTRAINT pk_PersonID PRIMARY KEY (Id_P,LastName)
)
```

SQL PRIMARY KEY Constraint on ALTER TABLE

如果在表已存在的情况下为 "Id_P" 列创建 PRIMARY KEY 约束，请使用下面的 SQL：

MySQL / SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
ADD PRIMARY KEY (Id_P)
```

如果需要命名 PRIMARY KEY 约束，以及为多个列定义 PRIMARY KEY 约束，请使用下面的 SQL 语法：

MySQL / SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
ADD CONSTRAINT pk_PersonID PRIMARY KEY (Id_P,LastName)
```

注释：如果您使用 ALTER TABLE 语句添加主键，必须把主键列声明为不包含 NULL 值（在表首次创建时）。

撤销 PRIMARY KEY 约束

如需撤销 PRIMARY KEY 约束，请使用下面的 SQL：

MySQL:

```
ALTER TABLE Persons
DROP PRIMARY KEY
```

SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
DROP CONSTRAINT pk_PersonID
```

SQL FOREIGN KEY 约束

SQL FOREIGN KEY 约束

一个表中的 FOREIGN KEY 指向另一个表中的 PRIMARY KEY。

让我们通过一个例子来解释外键。请看下面两个表：

"Persons" 表：

Id_P	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

"Orders" 表：

Id_O	OrderNo	Id_P
1	77895	3
2	44678	3
3	22456	1
4	24562	1

请注意，"Orders" 中的 "Id_P" 列指向 "Persons" 表中的 "Id_P" 列。

"Persons" 表中的 "Id_P" 列是 "Persons" 表中的 PRIMARY KEY。

"Orders" 表中的 "Id_P" 列是 "Orders" 表中的 FOREIGN KEY。

FOREIGN KEY 约束用于预防破坏表之间连接的动作。

FOREIGN KEY 约束也能防止非法数据插入外键列，因为它必须是它指向的那个表中的值之一。

SQL FOREIGN KEY Constraint on CREATE TABLE

下面的 SQL 在 "Orders" 表创建时为 "Id_P" 列创建 FOREIGN KEY：

MySQL:

```
CREATE TABLE Orders
(
Id_O int NOT NULL,
OrderNo int NOT NULL,
Id_P int,
PRIMARY KEY (Id_O),
FOREIGN KEY (Id_P) REFERENCES Persons(Id_P)
)
```

SQL Server / Oracle / MS Access:

```
CREATE TABLE Orders
(
Id_O int NOT NULL PRIMARY KEY,
OrderNo int NOT NULL,
Id_P int FOREIGN KEY REFERENCES Persons(Id_P)
)
```

如果需要命名 FOREIGN KEY 约束，以及为多个列定义 FOREIGN KEY 约束，请使用下面的 SQL 语法：

MySQL / SQL Server / Oracle / MS Access:

```
CREATE TABLE Orders
(
Id_O int NOT NULL,
OrderNo int NOT NULL,
Id_P int,
PRIMARY KEY (Id_O),
CONSTRAINT fk_PerOrders FOREIGN KEY (Id_P)
REFERENCES Persons(Id_P)
)
```

SQL FOREIGN KEY Constraint on ALTER TABLE

如果在 "Orders" 表已存在的情况下为 "Id_P" 列创建 FOREIGN KEY 约束，请使用下面的 SQL：

MySQL / SQL Server / Oracle / MS Access:

```
ALTER TABLE Orders
ADD FOREIGN KEY (Id_P)
REFERENCES Persons(Id_P)
```

如果需要命名 FOREIGN KEY 约束，以及为多个列定义 FOREIGN KEY 约束，请使用下面的 SQL 语法：

MySQL / SQL Server / Oracle / MS Access:

```
ALTER TABLE Orders
ADD CONSTRAINT fk_PerOrders
FOREIGN KEY (Id_P)
REFERENCES Persons(Id_P)
```

撤销 FOREIGN KEY 约束

如需撤销 FOREIGN KEY 约束，请使用下面的 SQL：

MySQL:

```
ALTER TABLE Orders
DROP FOREIGN KEY fk_PerOrders
```

SQL Server / Oracle / MS Access:

```
ALTER TABLE Orders
DROP CONSTRAINT fk_PerOrders
```

SQL CHECK 约束

SQL CHECK 约束

CHECK 约束用于限制列中的值的范围。

如果对单个列定义 CHECK 约束，那么该列只允许特定的值。

如果对一个表定义 CHECK 约束，那么此约束会在特定的列中对值进行限制。

SQL CHECK Constraint on CREATE TABLE

下面的 SQL 在 "Persons" 表创建时为 "Id_P" 列创建 CHECK 约束。CHECK 约束规定 "Id_P" 列必须只包含大于 0 的整数。

My SQL:

```
CREATE TABLE Persons
(
Id_P int NOT NULL,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255),
CHECK (Id_P>0)
)
```

SQL Server / Oracle / MS Access:

```
CREATE TABLE Persons
(
Id_P int NOT NULL CHECK (Id_P>0),
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255)
)
```

如果需要命名 CHECK 约束，以及为多个列定义 CHECK 约束，请使用下面的 SQL 语法：

MySQL / SQL Server / Oracle / MS Access:

```
CREATE TABLE Persons
(
Id_P int NOT NULL,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255),
CONSTRAINT chk_Person CHECK (Id_P>0 AND City='Sandnes')
)
```

SQL CHECK Constraint on ALTER TABLE

如果在表已存在的情况下为 "Id_P" 列创建 CHECK 约束，请使用下面的 SQL：

MySQL / SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
ADD CHECK (Id_P>0)
```

如果需要命名 CHECK 约束，以及为多个列定义 CHECK 约束，请使用下面的 SQL 语法：

MySQL / SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
ADD CONSTRAINT chk_Person CHECK (Id_P>0 AND City='Sandnes')
```

撤销 CHECK 约束

如需撤销 CHECK 约束，请使用下面的 SQL：

SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
DROP CONSTRAINT chk_Person
```

MySQL:

```
ALTER TABLE Persons
DROP CHECK chk_Person
```

SQL DEFAULT 约束

SQL DEFAULT 约束

DEFAULT 约束用于向列中插入默认值。

如果没有规定其他的值，那么会将默认值添加到所有的新记录。

SQL DEFAULT Constraint on CREATE TABLE

下面的 SQL 在 "Persons" 表创建时为 "City" 列创建 DEFAULT 约束：

My SQL / SQL Server / Oracle / MS Access:

```
CREATE TABLE Persons
(
Id_P int NOT NULL,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255) DEFAULT 'Sandnes'
)
```

通过使用类似 GETDATE() 这样的函数，DEFAULT 约束也可以用于插入系统值：

```
CREATE TABLE Orders
(
Id_O int NOT NULL,
OrderNo int NOT NULL,
Id_P int,
OrderDate date DEFAULT GETDATE()
)
```

SQL DEFAULT Constraint on ALTER TABLE

如果在表已存在的情况下为 "City" 列创建 DEFAULT 约束，请使用下面的 SQL：

MySQL:

```
ALTER TABLE Persons
ALTER City SET DEFAULT 'SANDNES'
```

SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
ALTER COLUMN City SET DEFAULT 'SANDNES'
```

撤销 DEFAULT 约束

如需撤销 DEFAULT 约束，请使用下面的 SQL：

MySQL:

```
ALTER TABLE Persons
ALTER City DROP DEFAULT
```

SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
ALTER COLUMN City DROP DEFAULT
```

SQL CREATE INDEX 语句

CREATE INDEX 语句用于在表中创建索引。

在不读取整个表的情况下，索引使数据库应用程序可以更快地查找数据。

索引

您可以在表中创建索引，以便更加快速高效地查询数据。

用户无法看到索引，它们只能被用来加速搜索/查询。

注释：更新一个包含索引的表需要比更新一个没有索引的表更多的时间，这是由于索引本身也需要更新。因此，理想的做法是仅仅在常常被搜索的列（以及表）上面创建索引。

SQL CREATE INDEX 语法

在表上创建一个简单的索引。允许使用重复的值：

```
CREATE INDEX index_name
ON table_name (column_name)
```

注释："column_name" 规定需要索引的列。

SQL CREATE UNIQUE INDEX 语法

在表上创建一个唯一的索引。唯一的索引意味着两个行不能拥有相同的索引值。

```
CREATE UNIQUE INDEX index_name
ON table_name (column_name)
```

CREATE INDEX 实例

本例会创建一个简单的索引，名为 "PersonIndex"，在 Person 表的 LastName 列：

```
CREATE INDEX PersonIndex
ON Person (LastName)
```

如果您希望以降序索引某个列中的值，您可以在列名称之后添加保留字 *DESC*：

```
CREATE INDEX PersonIndex
ON Person (LastName DESC)
```

假如您希望索引不止一个列，您可以在括号中列出这些列的名称，用逗号隔开：

```
CREATE INDEX PersonIndex
ON Person (LastName, FirstName)
```

SQL 撤销索引、表以及数据库

通过使用 DROP 语句，可以轻松地删除索引、表和数据库。

SQL DROP INDEX 语句

我们可以使用 DROP INDEX 命令删除表格中的索引。

用于 **Microsoft SQLJet (以及 Microsoft Access)** 的语法:

```
DROP INDEX index_name ON table_name
```

用于 **MS SQL Server** 的语法:

```
DROP INDEX table_name.index_name
```

用于 **IBM DB2** 和 **Oracle** 语法:

```
DROP INDEX index_name
```

用于 **MySQL** 的语法:

```
ALTER TABLE table_name DROP INDEX index_name
```

SQL DROP TABLE 语句

DROP TABLE 语句用于删除表（表的结构、属性以及索引也会被删除）：

```
DROP TABLE 表名称
```

SQL DROP DATABASE 语句

DROP DATABASE 语句用于删除数据库：

```
DROP DATABASE 数据库名称
```

SQL TRUNCATE TABLE 语句

如果我们仅仅需要除去表内的数据，但并不删除表本身，那么我们该如何做呢？

请使用 TRUNCATE TABLE 命令（仅仅删除表格中的数据）：

```
TRUNCATE TABLE 表名称
```

SQL ALTER TABLE 语句

ALTER TABLE 语句

ALTER TABLE 语句用于在已有的表中添加、修改或删除列。

SQL ALTER TABLE 语法

如需在表中添加列，请使用下列语法:

```
ALTER TABLE table_name
ADD column_name datatype
```

要删除表中的列，请使用下列语法：

```
ALTER TABLE table_name
DROP COLUMN column_name
```

注释：某些数据库系统不允许这种在数据库表中删除列的方式 (DROP COLUMN column_name)。

要改变表中列的数据类型，请使用下列语法：

```
ALTER TABLE table_name
ALTER COLUMN column_name datatype
```

原始的表 (用在例子中的)：

Persons 表:

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

SQL ALTER TABLE 实例

现在，我们希望在表 "Persons" 中添加一个名为 "Birthday" 的新列。

我们使用下列 SQL 语句：

```
ALTER TABLE Persons
ADD Birthday date
```

请注意，新列 "Birthday" 的类型是 date，可以存放日期。数据类型规定列中可以存放的数据的类型。

新的 "Persons" 表类似这样：

Id	LastName	FirstName	Address	City	Birthday
1	Adams	John	Oxford Street	London	
2	Bush	George	Fifth Avenue	New York	
3	Carter	Thomas	Changan Street	Beijing	

改变数据类型实例

现在我们希望改变 "Persons" 表中 "Birthday" 列的数据类型。

我们使用下列 SQL 语句：

```
ALTER TABLE Persons
ALTER COLUMN Birthday year
```

请注意，"Birthday" 列的数据类型是 year，可以存放 2 位或 4 位格式的年份。

DROP COLUMN 实例

接下来，我们删除 "Person" 表中的 "Birthday" 列：

```
ALTER TABLE Person
DROP COLUMN Birthday
```

Persons 表会成为这样:

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

SQL AUTO INCREMENT 字段

Auto-increment 会在新记录插入表中时生成一个唯一的数字。

AUTO INCREMENT 字段

我们通常希望在每次插入新记录时，自动地创建主键字段的值。

我们可以在表中创建一个 auto-increment 字段。

用于 MySQL 的语法

下列 SQL 语句把 "Persons" 表中的 "P_Id" 列定义为 auto-increment 主键：

```
CREATE TABLE Persons
(
P_Id int NOT NULL AUTO_INCREMENT,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255),
PRIMARY KEY (P_Id)
)
```

MySQL 使用 AUTO_INCREMENT 关键字来执行 auto-increment 任务。

默认地，AUTO_INCREMENT 的开始值是 1，每条新记录递增 1。

要让 AUTO_INCREMENT 序列以其他的值起始，请使用下列 SQL 语法：

```
ALTER TABLE Persons AUTO_INCREMENT=100
```

要在 "Persons" 表中插入新记录，我们不必为 "P_Id" 列规定值（会自动添加一个唯一的值）：

```
INSERT INTO Persons (FirstName,LastName)
VALUES ('Bill','Gates')
```

上面的 SQL 语句会在 "Persons" 表中插入一条新记录。"P_Id" 会被赋予一个唯一的值。"FirstName" 会被设置为 "Bill"，"LastName" 列会被设置为 "Gates"。

用于 **SQL Server** 的语法

下列 SQL 语句把 "Persons" 表中的 "P_Id" 列定义为 auto-increment 主键：

```
CREATE TABLE Persons
(
P_Id int PRIMARY KEY IDENTITY,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255)
)
```

MS SQL 使用 IDENTITY 关键字来执行 auto-increment 任务。

默认地，IDENTITY 的开始值是 1，每条新记录递增 1。

要规定 "P_Id" 列以 20 起始且递增 10，请把 identity 改为 IDENTITY(20,10)

要在 "Persons" 表中插入新记录，我们不必为 "P_Id" 列规定值（会自动添加一个唯一的值）：

```
INSERT INTO Persons (FirstName,LastName)
VALUES ('Bill','Gates')
```

上面的 SQL 语句会在 "Persons" 表中插入一条新记录。"P_Id" 会被赋予一个唯一的值。"FirstName" 会被设置为 "Bill"，"LastName" 列会被设置为 "Gates"。

用于 **Access** 的语法

下列 SQL 语句把 "Persons" 表中的 "P_Id" 列定义为 auto-increment 主键：

```
CREATE TABLE Persons
(
P_Id int PRIMARY KEY AUTOINCREMENT,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255)
)
```

MS Access 使用 AUTOINCREMENT 关键字来执行 auto-increment 任务。

默认地，AUTOINCREMENT 的开始值是 1，每条新记录递增 1。

要规定 "P_Id" 列以 20 起始且递增 10，请把 autoincrement 改为 AUTOINCREMENT(20,10)

要在 "Persons" 表中插入新记录，我们不必为 "P_Id" 列规定值（会自动添加一个唯一的值）：

```
INSERT INTO Persons (FirstName,LastName)
VALUES ('Bill','Gates')
```

上面的 SQL 语句会在 "Persons" 表中插入一条新记录。"P_Id" 会被赋予一个唯一的值。"FirstName" 会被设置为 "Bill"，"LastName" 列会被设置为 "Gates"。

用于 **Oracle** 的语法

在 Oracle 中，代码稍微复杂一点。

您必须通过 sequence 对创建 auto-increment 字段（该对象生成数字序列）。

请使用下面的 CREATE SEQUENCE 语法：

```
CREATE SEQUENCE seq_person
MINVALUE 1
START WITH 1
INCREMENT BY 1
CACHE 10
```

上面的代码创建名为 seq_person 的序列对象，它以 1 起始且以 1 递增。该对象缓存 10 个值以提高性能。CACHE 选项规定了为了提高访问速度要存储多少个序列值。

要在 "Persons" 表中插入新记录，我们必须使用 nextval 函数（该函数从 seq_person 序列中取回下一个值）：

```
INSERT INTO Persons (P_Id,FirstName,LastName)
VALUES (seq_person.nextval,'Lars','Monsen')
```

上面的 SQL 语句会在 "Persons" 表中插入一条新记录。"P_Id" 的赋值是来自 seq_person 序列的下一个数字。"FirstName" 会被设置为 "Bill"，"LastName" 列会被设置为 "Gates"。

SQL VIEW（视图）

视图是可视化的表。

本章讲解如何创建、更新和删除视图。

SQL CREATE VIEW 语句

什么是视图？

在 SQL 中，视图是基于 SQL 语句的结果集的可视化的表。

视图包含行和列，就像一个真实的表。视图中的字段就是来自一个或多个数据库中的真实的表中的字段。我们可以向视图添加 SQL 函数、WHERE 以及 JOIN 语句，我们也可以提交数据，就像这些来自于某个单一的表。

注释：数据库的设计和结构不会受到视图中的函数、where 或 join 语句的影响。

SQL CREATE VIEW 语法

```
CREATE VIEW view_name AS
SELECT column_name(s)
FROM table_name
WHERE condition
```

注释：视图总是显示最近的数据。每当用户查询视图时，数据库引擎通过使用 SQL 语句来重建数据。

SQL CREATE VIEW 实例

可以从某个查询内部、某个存储过程内部，或者从另一个视图内部来使用视图。通过向视图添加函数、join 等等，我们可以向用户精确地提交我们希望提交的数据。

样本数据库 Northwind 拥有一些被默认安装的视图。视图 "Current Product List" 会从 Products 表列出所有正在使用的产品。这个视图使用下列 SQL 创建：

```
CREATE VIEW [Current Product List] AS
SELECT ProductID,ProductName
FROM Products
WHERE Discontinued=No
```

我们可以查询上面这个视图：

```
SELECT * FROM [Current Product List]
```

Northwind 样本数据库的另一个视图会选取 Products 表中所有单位价格高于平均单位价格的产品：

```
CREATE VIEW [Products Above Average Price] AS
SELECT ProductName,UnitPrice
FROM Products
WHERE UnitPrice>(SELECT AVG(UnitPrice) FROM Products)
```

我们可以像这样查询上面这个视图：

```
SELECT * FROM [Products Above Average Price]
```

另一个来自 Northwind 数据库的视图实例会计算在 1997 年每个种类的销售总数。请注意，这个视图会从另一个名为 "Product Sales for 1997" 的视图那里选取数据：

```
CREATE VIEW [Category Sales For 1997] AS
SELECT DISTINCT CategoryName,Sum(ProductSales) AS CategorySales
FROM [Product Sales for 1997]
GROUP BY CategoryName
```

我们可以像这样查询上面这个视图：

```
SELECT * FROM [Category Sales For 1997]
```

我们也可以向查询添加条件。现在，我们仅仅需要查看 "Beverages" 类的全部销量：

```
SELECT * FROM [Category Sales For 1997]
WHERE CategoryName='Beverages'
```

SQL 更新视图

您可以使用下面的语法来更新视图：

```
SQL CREATE OR REPLACE VIEW Syntax
CREATE OR REPLACE VIEW view_name AS
SELECT column_name(s)
FROM table_name
WHERE condition
```

现在，我们希望向 "Current Product List" 视图添加 "Category" 列。我们将通过下列 SQL 更新视图：

```
CREATE VIEW [Current Product List] AS
SELECT ProductID,ProductName,Category
FROM Products
WHERE Discontinued=No
```

SQL 撤销视图

您可以通过 DROP VIEW 命令来删除视图。

```
SQL DROP VIEW Syntax
DROP VIEW view_name
```

SQL函数

SQL Date 函数

SQL 日期

当我们处理日期时，最难的任务恐怕是确保所插入的日期的格式，与数据库中日期列的格式相匹配。

只要数据包含的只是日期部分，运行查询就不会出问题。但是，如果涉及时间，情况就有点复杂了。

在讨论日期查询的复杂性之前，我们先来看看最重要的内建日期处理函数。

MySQL Date 函数

下面的表格列出了 MySQL 中最重要的内建日期函数：

函数	描述
NOW()	返回当前的日期和时间
CURDATE()	返回当前的日期
CURTIME()	返回当前的时间
DATE()	提取日期或日期/时间表达式的日期部分
EXTRACT()	返回日期/时间按的单独部分
DATE_ADD()	给日期添加指定的时间间隔
DATE_SUB()	从日期减去指定的时间间隔
DATEDIFF()	返回两个日期之间的天数
DATE_FORMAT()	用不同的格式显示日期/时间

SQL Server Date 函数

下面的表格列出了 SQL Server 中最重要的内建日期函数：

函数	描述
GETDATE()	返回当前日期和时间
DATEPART()	返回日期/时间的单独部分
DATEADD()	在日期中添加或减去指定的时间间隔
DATEDIFF()	返回两个日期之间的时间
CONVERT()	用不同的格式显示日期/时间

SQL Date 数据类型

MySQL 使用下列数据类型在数据库中存储日期或日期/时间值：

- DATE - 格式 YYYY-MM-DD
- DATETIME - 格式: YYYY-MM-DD HH:MM:SS
- TIMESTAMP - 格式: YYYY-MM-DD HH:MM:SS
- YEAR - 格式 YYYY 或 YY

SQL Server 使用下列数据类型在数据库中存储日期或日期/时间值：

- DATE - 格式 YYYY-MM-DD
- DATETIME - 格式: YYYY-MM-DD HH:MM:SS
- SMALLDATETIME - 格式: YYYY-MM-DD HH:MM:SS
- TIMESTAMP - 格式: 唯一的数字

SQL 日期处理

如果不涉及时间部分，那么我们可以轻松地比较两个日期！

假设我们有下面这个 "Orders" 表：

OrderId	ProductName	OrderDate
1	computer	2008-12-26
2	printer	2008-12-26
3	electrograph	2008-11-12
4	telephone	2008-10-19

现在，我们希望从上表中选取 OrderDate 为 "2008-12-26" 的记录。

我们使用如下 SELECT 语句：

```
SELECT * FROM Orders WHERE OrderDate='2008-12-26'
```

结果集：

OrderId	ProductName	OrderDate
1	computer	2008-12-26
3	electrograph	2008-12-26

现在假设 "Orders" 类似这样（请注意 "OrderDate" 列中的时间部分）：

OrderId	ProductName	OrderDate
1	computer	2008-12-26 16:23:55
2	printer	2008-12-26 10:45:26
3	electrograph	2008-11-12 14:12:08
4	telephone	2008-10-19 12:56:10

如果我们使用上面的 SELECT 语句：

```
SELECT * FROM Orders WHERE OrderDate='2008-12-26'
```

那么我们得不到结果。这是由于该查询不含有时间部分的日期。

提示：如果您希望使查询简单且更易维护，那么请不要在日期中使用时间部分！

SQL NULL 值

NULL 值是遗漏的未知数据。

默认地，表的列可以存放 **NULL** 值。

本章讲解 **IS NULL** 和 **IS NOT NULL** 操作符。

SQL NULL 值

如果表中的某个列是可选的，那么我们可以在不向该列添加值的情况下插入新记录或更新已有的记录。这意味着该字段将以 NULL 值保存。

NULL 值的处理方式与其他值不同。

NULL 用作未知的或不适用的值的占位符。

注释：无法比较 NULL 和 0；它们是不等价的。

SQL 的 NULL 值处理

请看下面的 "Persons" 表：

Id	LastName	FirstName	Address	City
1	Adams	John	London	
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Beijing	

假如 "Persons" 表中的 "Address" 列是可选的。这意味着如果在 "Address" 列插入一条不带值的记录，"Address" 列会使用 NULL 值保存。

那么我们如何测试 NULL 值呢？

无法使用比较运算符来测试 NULL 值，比如 =, <, 或者 <>。

我们必须使用 IS NULL 和 IS NOT NULL 操作符。

SQL IS NULL

我们如何仅仅选取在 "Address" 列中带有 NULL 值的记录呢？

我们必须使用 IS NULL 操作符：

```
SELECT LastName,FirstName,Address FROM Persons
WHERE Address IS NULL
```

结果集：

LastName	FirstName	Address
Adams	John	
Carter	Thomas	

提示：请始终使用 IS NULL 来查找 NULL 值。

SQL IS NOT NULL

我们如何选取在 "Address" 列中不带有 NULL 值的记录呢？

我们必须使用 IS NOT NULL 操作符：

```
SELECT LastName,FirstName,Address FROM Persons
WHERE Address IS NOT NULL
```

结果集：

LastName	FirstName	Address
Bush	George	Fifth Avenue

在下一节中，我们了解 ISNULL()、NVL()、IFNULL() 和 COALESCE() 函数。

SQL NULL 函数

SQL ISNULL()、NVL()、IFNULL() 和 COALESCE() 函数

请看下面的 "Products" 表：

P_Id	ProductName	UnitPrice	UnitsInStock	UnitsOnOrder
1	computer	699	25	15
2	printer	365	36	
3	telephone	280	159	57

假如 "UnitsOnOrder" 是可选的，而且可以包含 NULL 值。

我们使用如下 SELECT 语句：

```
SELECT ProductName,UnitPrice*(UnitsInStock+UnitsOnOrder)
FROM Products
```

在上面的例子中，如果有 "UnitsOnOrder" 值是 NULL，那么结果是 NULL。

微软的 ISNULL() 函数用于规定如何处理 NULL 值。

NVL(), IFNULL() 和 COALESCE() 函数也可以达到相同的结果。

在这里，我们希望 NULL 值为 0。

下面，如果 "UnitsOnOrder" 是 NULL，则不利于计算，因此如果值是 NULL 则 ISNULL() 返回 0。

SQL Server / MS Access

```
SELECT ProductName,UnitPrice*(UnitsInStock+ISNULL(UnitsOnOrder,0))
FROM Products
```

Oracle

Oracle 没有 ISNULL() 函数。不过，我们可以使用 NVL() 函数达到相同的结果：

```
SELECT ProductName,UnitPrice*(UnitsInStock+NVL(UnitsOnOrder,0))
FROM Products
```

MySQL

MySQL 也拥有类似 ISNULL() 的函数。不过它的工作方式与微软的 ISNULL() 函数有点不同。

在 MySQL 中，我们可以使用 IFNULL() 函数，就像这样：

```
SELECT ProductName,UnitPrice*(UnitsInStock+IFNULL(UnitsOnOrder,0))
FROM Products
```

或者我们可以使用 COALESCE() 函数，就像这样：

```
SELECT ProductName,UnitPrice*(UnitsInStock+COALESCE(UnitsOnOrder,0
FROM Products
```

SQL 数据类型

Microsoft Access、MySQL 以及 SQL Server 所使用的数据类型和范围。

Microsoft Access 数据类型

数据类型	描述	存储
Text	用于文本或文本与数字的组合。最多 255 个字符。	
Memo	Memo 用于更大数量的文本。最多存储 65,536 个字符。 注释：无法对 memo 字段进行排序。不过它们是可搜索的。	
Byte	允许 0 到 255 的数字。	1 字节
Integer	允许介于 -32,768 到 32,767 之间的数字。	2 字节
Long	允许介于 -2,147,483,648 与 2,147,483,647 之间的全部数字	4 字节
Single	单精度浮点。处理大多数小数。	4 字节
Double	双精度浮点。处理大多数小数。	8 字节
Currency	用于货币。支持 15 位的元，外加 4 位小数。 提示：您可以选择使用哪个国家的货币。	8 字节
AutoNumber	AutoNumber 字段自动为每条记录分配数字，通常从 1 开始。	4 字节
Date/Time	用于日期和时间	8 字节
Yes/No	逻辑字段，可以显示为 Yes/No、True/False 或 On/Off。 在代码中，使用常量 True 和 False （等价于 1 和 0） 注释：Yes/No 字段中不允许 Null 值	1 比特
Ole Object	可以存储图片、音频、视频或其他 BLOBs (Binary Large OBjects)	最多 1GB
Hyperlink	包含指向其他文件的链接，包括网页。	
Lookup Wizard	允许你创建一个可从下列列表中进行选择的选项列表。	4 字节

MySQL 数据类型

在 MySQL 中，有三种主要的类型：文本、数字和日期/时间类型。

Text 类型：

数据类型	描述
CHAR(size)	保存固定长度的字符串（可包含字母、数字以及特殊字符）。在括号中指定字符串的长度。最多 255 个字符。
VARCHAR(size)	保存可变长度的字符串（可包含字母、数字以及特殊字符）。在括号中指定字符串的最大长度。最多 255 个字符。 注释：如果值的长度大于 255，则被转换为 TEXT 类型。
TINYTEXT	存放最大长度为 255 个字符的字符串。
TEXT	存放最大长度为 65,535 个字符的字符串。
BLOB	用于 BLOBs (Binary Large OBjects)。存放最多 65,535 字节的数据。
MEDIUMTEXT	存放最大长度为 16,777,215 个字符的字符串。
MEDIUMBLOB	用于 BLOBs (Binary Large OBjects)。存放最多 16,777,215 字节的数据。
LONGTEXT	存放最大长度为 4,294,967,295 个字符的字符串。
LONGBLOB	用于 BLOBs (Binary Large OBjects)。存放最多 4,294,967,295 字节的数据。
ENUM(x,y,z,etc.)	允许你输入可能值的列表。可以在 ENUM 列表中列出最大 65535 个值。如果列表中不存在插入的值，则插入空值。注释：这些值是按照你输入的顺序存储的。 可以按照此格式输入可能的值：ENUM('X','Y','Z')
SET	与 ENUM 类似，SET 最多只能包含 64 个列表项，不过 SET 可存储一个以上的值。

Number 类型：

数据类型	描述
TINYINT(size)	-128 到 127 常规。0 到 255 无符号*。在括号中规定最大位数。
SMALLINT(size)	-32768 到 32767 常规。0 到 65535 无符号*。在括号中规定最大位数。
MEDIUMINT(size)	-8388608 到 8388607 普通。0 to 16777215 无符号*。在括号中规定最大位数。
INT(size)	-2147483648 到 2147483647 常规。0 到 4294967295 无符号*。在括号中规定最大位数。
BIGINT(size)	-9223372036854775808 到 9223372036854775807 常规。0 到 18446744073709551615 无符号*。在括号中规定最大位数。
FLOAT(size,d)	带有浮动小数点的小数字。在括号中规定最大位数。在 d 参数中规定小数点右侧的最大位数。
DOUBLE(size,d)	带有浮动小数点的大数字。在括号中规定最大位数。在 d 参数中规定小数点右侧的最大位数。
DECIMAL(size,d)	作为字符串存储的 DOUBLE 类型，允许固定的小数点。

- 这些整数类型拥有额外的选项 UNSIGNED。通常，整数可以是负数或正数。如果添加 UNSIGNED 属性，那么范围将从 0 开始，而不是某个负数。

Date 类型：

数据类型	描述
DATE()	日期。格式：YYYY-MM-DD 注释：支持的范围是从 '1000-01-01' 到 '9999-12-31'
DATETIME()	*日期和时间的组合。格式：YYYY-MM-DD HH:MM:SS 注释：支持的范围是从 '1000-01-01 00:00:00' 到 '9999-12-31 23:59:59'
TIMESTAMP()	*时间戳。TIMESTAMP 值使用 Unix 纪元('1970-01-01 00:00:00' UTC) 至今的描述来存储。格式：YYYY-MM-DD HH:MM:SS 注释：支持的范围是从 '1970-01-01 00:00:01' UTC 到 '2038-01-09 03:14:07' UTC
TIME()	时间。格式：HH:MM:SS 注释：支持的范围是从 '-838:59:59' 到 '838:59:59'
YEAR()	2 位或 4 位格式的年。 注释：4 位格式所允许的值：1901 到 2155。2 位格式所允许的值：70 到 69，表示从 1970 到 2069。

- 即便 DATETIME 和 TIMESTAMP 返回相同的格式，它们的工作方式很不同。在 INSERT 或 UPDATE 查询中，TIMESTAMP 自动把自身设置为当前的日期和时间。TIMESTAMP 也接受不同的格式，比如 YYYYMMDDHHMMSS、YYMMDDHHMMSS、YYYYMMDD 或 YYMMDD。

SQL Server 数据类型

Character 字符串：

数据类型	描述	存储
char(n)	固定长度的字符串。最多 8,000 个字符。	n
varchar(n)	可变长度的字符串。最多 8,000 个字符。	
varchar(max)	可变长度的字符串。最多 1,073,741,824 个字符。	
text	可变长度的字符串。最多 2GB 字符数据。	

Unicode 字符串：

数据类型	描述	存储
nchar(n)	固定长度的 Unicode 数据。最多 4,000 个字符。	
nvarchar(n)	可变长度的 Unicode 数据。最多 4,000 个字符。	
nvarchar(max)	可变长度的 Unicode 数据。最多 536,870,912 个字符。	
ntext	可变长度的 Unicode 数据。最多 2GB 字符数据。	

Binary 类型：

数据类型	描述	存储
bit	允许 0、1 或 NULL	
binary(n)	固定长度的二进制数据。最多 8,000 字节。	
varbinary(n)	可变长度的二进制数据。最多 8,000 字节。	
varbinary(max)	可变长度的二进制数据。最多 2GB 字节。	
image	可变长度的二进制数据。最多 2GB。	

Number 类型：

数据类型	描述	存储
tinyint	允许从 0 到 255 的所有数字。	1 字节
smallint	允许从 -32,768 到 32,767 的所有数字。	2 字节
int	允许从 -2,147,483,648 到 2,147,483,647 的所有数字。	4 字节
bigint	允许介于 -9,223,372,036,854,775,808 和 9,223,372,036,854,775,807 之间的所有数字。	8 字节
decimal(p,s)	固定精度和比例的数字。允许从 -10^38 +1 到 10^38 -1 之间的数字。 p 参数指示可以存储的最大位数（小数点左侧和右侧）。p 必须是 1 到 38 之间的值。默认是 18。 s 参数指示小数点右侧存储的最大位数。s 必须是 0 到 p 之间的值。默认是 0。	5-17 字节
numeric(p,s)	固定精度和比例的数字。允许从 -10^38 +1 到 10^38 -1 之间的数字。 p 参数指示可以存储的最大位数（小数点左侧和右侧）。p 必须是 1 到 38 之间的值。默认是 18。 s 参数指示小数点右侧存储的最大位数。s 必须是 0 到 p 之间的值。默认是 0。	5-17 字节
smallmoney	介于 -214,748.3648 和 214,748.3647 之间的货币数据。	4 字节
money	介于 -922,337,203,685,477.5808 和 922,337,203,685,477.5807 之间的货币数据。	8 字节
float(n)	从 -1.79E + 308 到 1.79E + 308 的浮动精度数字数据。 参数 n 指示该字段保存 4 字节还是 8 字节。float(24) 保存 4 字节，而 float(53) 保存 8 字节。n 的默认值是 53。	4 或 8 字节
real	从 -3.40E + 38 到 3.40E + 38 的浮动精度数字数据。	4 字节

Date 类型：

数据类型	描述	存储
datetime	从 1753 年 1 月 1 日 到 9999 年 12 月 31 日，精度为 3.33 毫秒。	8 bytes
datetime2	从 1753 年 1 月 1 日 到 9999 年 12 月 31 日，精度为 100 纳秒。	6-8 bytes
smalldatetime	从 1900 年 1 月 1 日 到 2079 年 6 月 6 日，精度为 1 分钟。	4 bytes
date	仅存储日期。从 0001 年 1 月 1 日 到 9999 年 12 月 31 日。	3 bytes
time	仅存储时间。精度为 100 纳秒。	3-5 bytes
datetimeoffset	与 datetime2 相同，外加时区偏移。	8-10 bytes
timestamp	存储唯一的数字，每当创建或修改某行时，该数字会更新。timestamp 基于内部时钟，不对应真实时间。每个表只能有一个 timestamp 变量。	

其他数据类型：

数据类型	描述
sql_variant	存储最多 8,000 字节不同数据类型的数据，除了 text、ntext 以及 timestamp。
uniqueidentifier	存储全局标识符 (GUID)。
xml	存储 XML 格式化数据。最多 2GB。
cursor	存储对用于数据库操作的指针的引用。
table	存储结果集，供稍后处理。

SQL 服务器 - RDBMS

现代的 **SQL** 服务器构建在 **RDBMS** 之上。

DBMS - 数据库管理系统（Database Management System）

数据库管理系统是一种可以访问数据库中数据的计算机程序。

DBMS 使我们有能力在数据库中提取、修改或者存贮信息。

不同的 DBMS 提供不同的函数供查询、提交以及修改数据。

RDBMS - 关系数据库管理系统（Relational Database Management System）

关系数据库管理系统 (RDBMS) 也是一种数据库管理系统，其数据库是根据数据间的关系来组织和访问数据的。

20 世纪 70 年代初，IBM 公司发明了 RDBMS。

RDBMS 是 SQL 的基础，也是所有现代数据库系统诸如 Oracle、SQL Server、IBM DB2、Sybase、MySQL 以及 Microsoft Access 的基础。

SQL 函数

SQL 拥有很多可用于计数和计算的内建函数。

函数的语法

内建 SQL 函数的语法是：

```
SELECT function(列) FROM 表
```

函数的类型

在 SQL 中，基本的函数类型和种类有若干种。函数的基本类型是：

- Aggregate 函数
- Scalar 函数

合计函数（**Aggregate functions**）

Aggregate 函数的操作面向一系列的值，并返回一个单一的值。

注释：如果在 SELECT 语句的项目列表中的众多其它表达式中使用 SELECT 语句，则这个 SELECT 必须使用 GROUP BY 语句！

"Persons" table (在大部分的例子中使用过)

Name	Age
Adams, John	38
Bush, George	33
Carter, Thomas	28

MS Access 中的合计函数

函数	描述
AVG(column)	返回某列的平均值
COUNT(column)	返回某列的行数（不包括 NULL 值）
COUNT(*)	返回被选行数
FIRST(column)	返回在指定的域中第一个记录的值
LAST(column)	返回在指定的域中最后一个记录的值
MAX(column)	返回某列的最高值
MIN(column)	返回某列的最低值
STDEV(column)	
STDEVP(column)	
SUM(column)	返回某列的总和
VAR(column)	
VARP(column)	

在 SQL Server 中的合计函数

函数	描述
AVG(column)	返回某列的平均值
BINARY_CHECKSUM	
CHECKSUM	
CHECKSUM_AGG	
COUNT(column)	返回某列的行数（不包括NULL值）
COUNT(*)	返回被选行数
COUNT(DISTINCT column)	返回相异结果的数目
FIRST(column)	返回在指定的域中第一个记录的值（SQLServer2000 不支持）
LAST(column)	返回在指定的域中最后一个记录的值（SQLServer2000 不支持）
MAX(column)	返回某列的最高值
MIN(column)	返回某列的最低值
STDEV(column)	
STDEVP(column)	
SUM(column)	返回某列的总和
VAR(column)	
VARP(column)	

Scalar 函数 Scalar 函数的操作面向某个单一的值，并返回基于输入值的一个单一的值。 ### MS Access 中的 Scalar 函数

函数	描述
UCASE(c)	将某个域转换为大写
LCASE(c)	将某个域转换为小写
MID(c,start[,end])	从某个文本域提取字符
LEN(c)	返回某个文本域的长度
INSTR(c,char)	返回在某个文本域中指定字符的数值位置
LEFT(c,number_of_char)	返回某个被请求的文本域的左侧部分
RIGHT(c,number_of_char)	返回某个被请求的文本域的右侧部分
ROUND(c,decimals)	对某个数值域进行指定小数位数的四舍五入
MOD(x,y)	返回除法操作的余数
NOW()	返回当前的系统日期
FORMAT(c,format)	改变某个域的显示方式
DATEDIFF(d,date1,date2)	用于执行日期计算

SQL AVG 函数

定义和用法

AVG 函数返回数值列的平均值。NULL 值不包括在计算中。

SQL AVG() 语法

```
SELECT AVG(column_name) FROM table_name
```

SQL AVG() 实例

我们拥有下面这个 "Orders" 表：

O_Id	OrderDate	OrderPrice	Customer
1	2008/12/29	1000	Bush
2	2008/11/23	1600	Carter
3	2008/10/05	700	Bush
4	2008/09/28	300	Bush
5	2008/08/06	2000	Adams
6	2008/07/21	100	Carter

例子 1

现在，我们希望计算 "OrderPrice" 字段的平均值。

我们使用如下 SQL 语句：

```
SELECT AVG(OrderPrice) AS OrderAverage FROM Orders
```

结果集类似这样：

OrderAverage
950

例子 2

现在，我们希望找到 OrderPrice 值高于 OrderPrice 平均值的客户。

我们使用如下 SQL 语句：

```
SELECT Customer FROM Orders
WHERE OrderPrice>(SELECT AVG(OrderPrice) FROM Orders)
```

结果集类似这样：

Customer
Bush
Carter
Adams

SQL COUNT() 函数

COUNT() 函数返回匹配指定条件的行数。

SQL COUNT() 语法

SQL COUNT(column_name) 语法

COUNT(column_name) 函数返回指定列的值的数目（NULL 不计入）：

```
SELECT COUNT(column_name) FROM table_name
```

SQL COUNT(*) 语法

COUNT(*) 函数返回表中的记录数：

```
SELECT COUNT(*) FROM table_name
```

SQL COUNT(DISTINCT column_name) 语法

COUNT(DISTINCT column_name) 函数返回指定列的不同值的数目：

```
SELECT COUNT(DISTINCT column_name) FROM table_name
```

注释：COUNT(DISTINCT) 适用于 ORACLE 和 Microsoft SQL Server，但是无法用于 Microsoft Access。

SQL COUNT(column_name) 实例

我们拥有下列 "Orders" 表：

O_Id	OrderDate	OrderPrice	Customer
1	2008/12/29	1000	Bush
2	2008/11/23	1600	Carter
3	2008/10/05	700	Bush
4	2008/09/28	300	Bush
5	2008/08/06	2000	Adams
6	2008/07/21	100	Carter

现在，我们希望计算客户 "Carter" 的订单数。

我们使用如下 SQL 语句：

```
SELECT COUNT(Customer) AS CustomerNilsen FROM Orders
WHERE Customer='Carter'
```

以上 SQL 语句的结果是 2，因为客户 Carter 共有 2 个订单：

CustomerNilsen
2

SQL COUNT(*) 实例

如果我们省略 WHERE 子句，比如这样：

```
SELECT COUNT(*) AS NumberOfOrders FROM Orders
```

结果集类似这样：

NumberOfOrders
6

这是表中的总行数。

SQL COUNT(DISTINCT column_name) 实例

现在，我们希望计算 "Orders" 表中不同客户的数目。

我们使用如下 SQL 语句：

```
SELECT COUNT(DISTINCT Customer) AS NumberOfCustomers FROM Orders
```

结果集类似这样：

NumberOfCustomers
3

这是 "Orders" 表中不同客户（Bush, Carter 和 Adams）的数目。

SQL FIRST() 函数

FIRST() 函数

FIRST() 函数返回指定的字段中第一个记录的值。

提示：可使用 ORDER BY 语句对记录进行排序。

SQL FIRST() 语法

```
SELECT FIRST(column_name) FROM table_name
```

SQL FIRST() 实例

我们拥有下面这个 "Orders" 表：

O_Id	OrderDate	OrderPrice	Customer
1	2008/12/29	1000	Bush
2	2008/11/23	1600	Carter
3	2008/10/05	700	Bush
4	2008/09/28	300	Bush
5	2008/08/06	2000	Adams
6	2008/07/21	100	Carter

现在，我们希望查找 "OrderPrice" 列的第一个值。

我们使用如下 SQL 语句：

```
SELECT FIRST(OrderPrice) AS FirstOrderPrice FROM Orders
```

结果集类似这样：

FirstOrderPrice
1000

SQL LAST() 函数

LAST() 函数

LAST() 函数返回指定的字段中最后一个记录的值。

提示：可使用 ORDER BY 语句对记录进行排序。

SQL LAST() 语法

```
SELECT LAST(column_name) FROM table_name
```

SQL LAST() 实例

我们拥有下面这个 "Orders" 表：

O_Id	OrderDate	OrderPrice	Customer
1	2008/12/29	1000	Bush
2	2008/11/23	1600	Carter
3	2008/10/05	700	Bush
4	2008/09/28	300	Bush
5	2008/08/06	2000	Adams
6	2008/07/21	100	Carter

现在，我们希望查找 "OrderPrice" 列的最后一个值。

我们使用如下 SQL 语句：

```
SELECT LAST(OrderPrice) AS LastOrderPrice FROM Orders
```

结果集类似这样：

LastOrderPrice
100

SQL MAX() 函数

MAX() 函数

MAX 函数返回一列中的最大值。NULL 值不包括在计算中。

SQL MAX() 语法

```
SELECT MAX(column_name) FROM table_name
```

注释：MIN 和 MAX 也可用于文本列，以获得按字母顺序排列的最高或最低值。

SQL MAX() 实例

我们拥有下面这个 "Orders" 表：

O_Id	OrderDate	OrderPrice	Customer
1	2008/12/29	1000	Bush
2	2008/11/23	1600	Carter
3	2008/10/05	700	Bush
4	2008/09/28	300	Bush
5	2008/08/06	2000	Adams
6	2008/07/21	100	Carter

现在，我们希望查找 "OrderPrice" 列的最大值。

我们使用如下 SQL 语句：

```
SELECT MAX(OrderPrice) AS LargestOrderPrice FROM Orders
```

结果集类似这样：

LargestOrderPrice
2000

SQL MIN() 函数

MIN() 函数

MIN 函数返回一列中的最小值。NULL 值不包括在计算中。

SQL MIN() 语法

```
SELECT MIN(column_name) FROM table_name
```

注释：MIN 和 MAX 也可用于文本列，以获得按字母顺序排列的最高或最低值。

SQL MIN() 实例

我们拥有下面这个 "Orders" 表：

O_Id	OrderDate	OrderPrice	Customer
1	2008/12/29	1000	Bush
2	2008/11/23	1600	Carter
3	2008/10/05	700	Bush
4	2008/09/28	300	Bush
5	2008/08/06	2000	Adams
6	2008/07/21	100	Carter

现在，我们希望查找 "OrderPrice" 列的最小值。

我们使用如下 SQL 语句：

```
SELECT MIN(OrderPrice) AS SmallestOrderPrice FROM Orders
```

结果集类似这样：

SmallestOrderPrice
100

SQL SUM() 函数

SUM() 函数

SUM 函数返回数值列的总数（总额）。

SQL SUM() 语法

```
SELECT SUM(column_name) FROM table_name
```

SQL SUM() 实例

我们拥有下面这个 "Orders" 表：

O_Id	OrderDate	OrderPrice	Customer
1	2008/12/29	1000	Bush
2	2008/11/23	1600	Carter
3	2008/10/05	700	Bush
4	2008/09/28	300	Bush
5	2008/08/06	2000	Adams
6	2008/07/21	100	Carter

现在，我们希望查找 "OrderPrice" 字段的总数。

我们使用如下 SQL 语句：

```
SELECT SUM(OrderPrice) AS OrderTotal FROM Orders
```

结果集类似这样：

OrderTotal
5700

SQL GROUP BY 语句

合计函数 (比如 **SUM**) 常常需要添加 **GROUP BY** 语句。

GROUP BY 语句

GROUP BY 语句用于结合合计函数，根据一个或多个列对结果集进行分组。

SQL GROUP BY 语法

```
SELECT column_name, aggregate_function(column_name)
FROM table_name
WHERE column_name operator value
GROUP BY column_name
```

SQL GROUP BY 实例

我们拥有下面这个 "Orders" 表：

O_Id	OrderDate	OrderPrice	Customer
1	2008/12/29	1000	Bush
2	2008/11/23	1600	Carter
3	2008/10/05	700	Bush
4	2008/09/28	300	Bush
5	2008/08/06	2000	Adams
6	2008/07/21	100	Carter

现在，我们希望查找每个客户的总金额（总订单）。

我们想要使用 GROUP BY 语句对客户进行组合。

我们使用下列 SQL 语句：

```
SELECT Customer,SUM(OrderPrice) FROM Orders
GROUP BY Customer
```

结果集类似这样：

Customer	SUM(OrderPrice)
Bush	2000
Carter	1700
Adams	2000

很棒吧，对不对？

让我们看一下如果省略 GROUP BY 会出现什么情况：

```
SELECT Customer,SUM(OrderPrice) FROM Orders
```

结果集类似这样：

Customer	SUM(OrderPrice)
Bush	5700
Carter	5700
Bush	5700
Bush	5700
Adams	5700
Carter	5700

上面的结果集不是我们需要的。

那么为什么不能使用上面这条 SELECT 语句呢？解释如下：上面的 SELECT 语句指定了两列（Customer 和 SUM(OrderPrice)）。"SUM(OrderPrice)" 返回一个单独的值（"OrderPrice" 列的总计），而 "Customer" 返回 6 个值（每个值对应 "Orders" 表中的每一行）。因此，我们得不到正确的结果。不过，您已经看到了，GROUP BY 语句解决了这个问题。

GROUP BY 一个以上的列

我们也可以对一个以上的列应用 GROUP BY 语句，就像这样：

```
SELECT Customer,OrderDate,SUM(OrderPrice) FROM Orders
GROUP BY Customer,OrderDate
```

SQL HAVING 子句

HAVING 子句

在 SQL 中增加 HAVING 子句原因是，WHERE 关键字无法与合计函数一起使用。

SQL HAVING 语法

```
SELECT column_name, aggregate_function(column_name)
FROM table_name
WHERE column_name operator value
GROUP BY column_name
HAVING aggregate_function(column_name) operator value
```

SQL HAVING 实例

我们拥有下面这个 "Orders" 表：

O_Id	OrderDate	OrderPrice	Customer
1	2008/12/29	1000	Bush
2	2008/11/23	1600	Carter
3	2008/10/05	700	Bush
4	2008/09/28	300	Bush
5	2008/08/06	2000	Adams
6	2008/07/21	100	Carter

现在，我们希望查找订单总金额少于 2000 的客户。

我们使用如下 SQL 语句：

```
SELECT Customer,SUM(OrderPrice) FROM Orders
GROUP BY Customer
HAVING SUM(OrderPrice)<2000
```

结果集类似：

Customer	SUM(OrderPrice)
Carter	1700

现在我们希望查找客户 "Bush" 或 "Adams" 拥有超过 1500 的订单总金额。

我们在 SQL 语句中增加了一个普通的 WHERE 子句：

```
SELECT Customer,SUM(OrderPrice) FROM Orders
WHERE Customer='Bush' OR Customer='Adams'
GROUP BY Customer
HAVING SUM(OrderPrice)>1500
```

结果集：

Customer	SUM(OrderPrice)
Bush	2000
Adams	2000

SQL UCASE() 函数

UCASE() 函数

UCASE 函数把字段的值转换为大写。

SQL UCASE() 语法

```
SELECT UCASE(column_name) FROM table_name
```

SQL UCASE() 实例

我们拥有下面这个 "Persons" 表：

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

现在，我们希望选取 "LastName" 和 "FirstName" 列的内容，然后把 "LastName" 列转换为大写。

我们使用如下 SQL 语句：

```
SELECT UCASE(LastName) as LastName,FirstName FROM Persons
```

结果集类似这样：

LastName	FirstName
ADAMS	John
BUSH	George
CARTER	Thomas

SQL LCASE() 函数

LCASE() 函数

LCASE 函数把字段的值转换为小写。

SQL LCASE() 语法

```
SELECT LCASE(column_name) FROM table_name
```

SQL LCASE() 实例

我们拥有下面这个 "Persons" 表：

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

现在，我们希望选取 "LastName" 和 "FirstName" 列的内容，然后把 "LastName" 列转换为小写。

我们使用如下 SQL 语句：

```
SELECT LCASE(LastName) as LastName,FirstName FROM Persons
```

结果集类似这样：

LastName	FirstName
adams	John
bush	George
carter	Thomas

SQL MID() 函数

MID() 函数

MID 函数用于从文本字段中提取字符。

SQL MID() 语法

```
SELECT MID(column_name,start[,length]) FROM table_name
```

参数	描述
column_name	必需。要提取字符的字段。
start	必需。规定开始位置（起始值是 1）。
length	可选。要返回的字符数。如果省略，则 MID() 函数返回剩余文本。

SQL MID() 实例

我们拥有下面这个 "Persons" 表：

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

现在，我们希望从 "City" 列中提取前 3 个字符。

我们使用如下 SQL 语句：

```
SELECT MID(City,1,3) as SmallCity FROM Persons
```

结果集类似这样：

SmallCity
Lon
New
Bei

SQL LEN() 函数

LEN() 函数

LEN 函数返回文本字段中值的长度。

SQL LEN() 语法

```
SELECT LEN(column_name) FROM table_name
```

SQL LEN() 实例

我们拥有下面这个 "Persons" 表：

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

现在，我们希望取得 "City" 列中值的长度。

我们使用如下 SQL 语句：

```
SELECT LEN(City) as LengthOfCity FROM Persons
```

结果集类似这样：

LengthOfCity
6
8
7

SQL ROUND() 函数

ROUND() 函数

ROUND 函数用于把数值字段舍入为指定的小数位数。

SQL ROUND() 语法

```
SELECT ROUND(column_name,decimals) FROM table_name
```

参数	描述
column_name	必需。要舍入的字段。
decimals	必需。规定要返回的小数位数。

SQL ROUND() 实例

我们拥有下面这个 "Products" 表：

Prod_Id	ProductName	Unit	UnitPrice
1	gold	1000 g	32.35
2	silver	1000 g	11.56
3	copper	1000 g	6.85

现在，我们希望把名称和价格舍入为最接近的整数。

我们使用如下 SQL 语句：

```
SELECT ProductName, ROUND(UnitPrice,0) as UnitPrice FROM Products
```

结果集类似这样：

ProductName	UnitPrice
gold	32
silver	12
copper	7

SQL NOW() 函数

NOW() 函数

NOW 函数返回当前的日期和时间。

提示：如果您在使用 Sql Server 数据库，请使用 getdate() 函数来获得当前的日期时间。

SQL NOW() 语法

```
SELECT NOW() FROM table_name
```

SQL NOW() 实例

我们拥有下面这个 "Products" 表：

Prod_Id	ProductName	Unit	UnitPrice
1	gold	1000 g	32.35
2	silver	1000 g	11.56
3	copper	1000 g	6.85

现在，我们希望显示当天的日期所对应的名称和价格。

我们使用如下 SQL 语句：

```
SELECT ProductName, UnitPrice, Now() as PerDate FROM Products
```

结果集类似这样：

ProductName	UnitPrice	PerDate
gold	32.35	12/29/2008 11:36:05 AM
silver	11.56	12/29/2008 11:36:05 AM
copper	6.85	12/29/2008 11:36:05 AM

SQL FORMAT() 函数

FORMAT() 函数

FORMAT 函数用于对字段的显示进行格式化。

SQL FORMAT() 语法

```
SELECT FORMAT(column_name,format) FROM table_name
```

参数	描述
column_name	必需。要格式化的字段。
format	必需。规定格式。

SQL FORMAT() 实例

我们拥有下面这个 "Products" 表：

Prod_Id	ProductName	Unit	UnitPrice
1	gold	1000 g	32.35
2	silver	1000 g	11.56
3	copper	1000 g	6.85

现在，我们希望显示每天日期所对应的名称和价格（日期的显示格式是 "YYYY-MM-DD"）。

我们使用如下 SQL 语句：

```
SELECT ProductName, UnitPrice, FORMAT(Now(),'YYYY-MM-DD') as PerDat
FROM Products
```

结果集类似这样：

ProductName	UnitPrice	PerDate
gold	32.35	12/29/2008
silver	11.56	12/29/2008
copper	6.85	12/29/2008

SQL 快速参考

来自 **W3School** 的 **SQL** 快速参考。可以打印它，以备日常使用。

SQL 语句

语句	语法
AND / OR	SELECT column_name(s) FROM table_name WHERE condition AND|OR condition
ALTER TABLE (add column)	ALTER TABLE table_name ADD column_name datatype
ALTER TABLE (drop column)	ALTER TABLE table_name DROP COLUMN column_name
AS (alias for column)	SELECT column_name AS column_alias FROM table_name
AS (alias for table)	SELECT column_name FROM table_name AS table_alias
BETWEEN	SELECT column_name(s) FROM table_name WHERE column_name BETWEEN value1 AND value2
CREATE DATABASE	CREATE DATABASE database_name
CREATE INDEX	CREATE INDEX index_name ON table_name (column_name)
CREATE TABLE	CREATE TABLE table_name (column_name1 data_type, column_name2 data_type, ...)
CREATE UNIQUE INDEX	CREATE UNIQUE INDEX index_name ON table_name (column_name)
CREATE VIEW	CREATE VIEW view_name AS SELECT column_name(s) FROM table_name WHERE condition
DELETE FROM	DELETE FROM table*name (**Note:** Deletes the entire table!!) _or* DELETE FROM table_name WHERE condition
DROP DATABASE	DROP DATABASE database_name
DROP INDEX	DROP INDEX table_name.index_name
DROP TABLE	DROP TABLE table_name

GROUP BY	SELECT column_name1,SUM(column_name2) FROM table_name GROUP BY column_name1
HAVING	SELECT column_name1,SUM(column_name2) FROM table_name GROUP BY column_name1 HAVING SUM(column_name2) condition value
IN	SELECT column_name(s) FROM table_name WHERE column_name IN (value1,value2,..)
INSERT INTO	INSERT INTO table*name VALUES (value1, value2,....) _or* INSERT INTO table_name (column_name1, column_name2,...) VALUES (value1, value2,....)
LIKE	SELECT column_name(s) FROM table_name WHERE column_name LIKE pattern
ORDER BY	SELECT column_name(s) FROM table_name ORDER BY column_name [ASC|DESC]
SELECT	SELECT column_name(s) FROM table_name
SELECT *	SELECT * FROM table_name
SELECT DISTINCT	SELECT DISTINCT column_name(s) FROM table_name
SELECT INTO (used to create backup copies of tables)	SELECT * INTO new*table_name FROM original_table_name _or* SELECT column_name(s) INTO new_table_name FROM original_table_name
TRUNCATE TABLE (deletes only the data inside the table)	TRUNCATE TABLE table_name
UPDATE	UPDATE table_name SET column_name=new_value [, column_name=new_value] WHERE column_name=some_value
WHERE	SELECT column_name(s) FROM table_name WHERE condition

SQLite 基础

SQLite 简介

本教程帮助您了解什么是 SQLite，它与 SQL 之间的不同，为什么需要它，以及它的应用程序数据库处理方式。

SQLite是一个软件库，实现了自给自足的、无服务器的、零配置的、事务性的 SQL 数据库引擎。SQLite是一个增长最快的数据库引擎，这是在普及方面的增长，与它的尺寸大小无关。SQLite 源代码不受版权限制。

什么是 SQLite？

SQLite是一个进程内的库，实现了自给自足的、无服务器的、零配置的、事务性的 SQL 数据库引擎。它是一个零配置的数据库，这意味着与其他数据库一样，您不需要在系统中配置。

就像其他数据库，SQLite 引擎不是一个独立的进程，可以按应用程序需求进行静态或动态连接。SQLite 直接访问其存储文件。

为什么要用 SQLite？

- 不需要一个单独的服务器进程或操作的系统（无服务器的）。
- SQLite 不需要配置，这意味着不需要安装或管理。
- 一个完整的 SQLite 数据库是存储在一个单一的跨平台的磁盘文件。
- SQLite 是非常小的，是轻量级的，完全配置时小于 400KiB，省略可选功能配置时小于250KiB。
- SQLite 是自给自足的，这意味着不需要任何外部的依赖。
- SQLite 事务是完全兼容 ACID 的，允许从多个进程或线程安全访问。
- SQLite 支持 SQL92（SQL2）标准的大多数查询语言的功能。
- SQLite 使用 ANSI-C 编写的，并提供了简单和易于使用的 API。
- SQLite 可在 UNIX（Linux, Mac OS-X, Android, iOS）和 Windows（Win32, WinCE, WinRT）中运行。

历史

1. 2000 -- D. Richard Hipp 设计 SQLite 是为了不需要管理即可操作程序。
2. 2000 -- 在八月，SQLite1.0 发布 GNU 数据库管理器（GNU Database Manager）。

3. 2011 -- Hipp 宣布，向 SQLite DB 添加 UNQl 接口，开发 UNQLite（面向文档的数据库）。

SQLite 局限性

在 SQLite 中，SQL92 不支持的特性如下所示：

特性	描述
RIGHT OUTER JOIN	只实现了 LEFT OUTER JOIN。
FULL OUTER JOIN	只实现了 LEFT OUTER JOIN。
ALTER TABLE	支持 RENAME TABLE 和 ALTER TABLE 的 ADD COLUMN variants 命令，不支持 DROP COLUMN、ALTER COLUMN、ADD CONSTRAINT。
Trigger 支持	支持 FOR EACH ROW 触发器，但不支持 FOR EACH STATEMENT 触发器。
VIEWs	在 SQLite 中，视图是只读的。您不可以在视图上执行 DELETE、INSERT 或 UPDATE 语句。
GRANT 和 REVOKE	可以应用的唯一的访问权限是底层操作系统的正常文件访问权限。

SQLite 命令

与关系数据库进行交互的标准 SQLite 命令类似于 SQL。命令包括 CREATE、SELECT、INSERT、UPDATE、DELETE 和 DROP。这些命令基于它们的操作性质可分为以下几种：

DDL - 数据定义语言

命令	描述
CREATE	创建一个新的表，一个表的视图，或者数据库中的其他对象。
ALTER	修改数据库中的某个已有的数据库对象，比如一个表。
DROP	删除整个表，或者表的视图，或者数据库中的其他对象。

DML - 数据操作语言

命令	描述
INSERT	创建一条记录。
UPDATE	修改记录。
DELETE	删除记录。

DQL - 数据查询语言

命令	描述
SELECT	从一个或多个表中检索某些记录。

SQLite 安装

SQLite 的一个重要的特性是零配置的，这意味着不需要复杂的安装或管理。本章将讲解 Windows、Linux 和 Mac OS X 上的安装设置。

在 Windows 上安装 SQLite

- 请访问 SQLite 下载页面，从 Windows 区下载预编译的二进制文件。
- 您需要下载 **sqlite-shell-win32-*.zip** 和 **sqlite-dll-win32-*.zip** 压缩文件。
- 创建文件夹 C:\>sqlite，并在此文件夹下解压上面两个压缩文件，将得到 sqlite3.def、sqlite3.dll 和 sqlite3.exe 文件。
- 添加 C:\>sqlite 到 PATH 环境变量，最后在命令提示符下，使用 **sqlite3** 命令，将显示如下结果。

```
C:\>sqlite3
SQLite version 3.7.15.2 2013-01-09 11:53:05
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

在 Linux 上安装 SQLite

目前，几乎所有版本的 Linux 操作系统都附带 SQLite。所以，只要使用下面的命令来检查您的机器上是否已经安装了 SQLite。

```
$sqlite3
SQLite version 3.7.15.2 2013-01-09 11:53:05
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

如果没有看到上面的结果，那么就意味着没有在 Linux 机器上安装 SQLite。因此，让我们按照下面的步骤安装 SQLite：

- 请访问 SQLite 下载页面，从源代码区下载 **sqlite-autoconf-*.tar.gz**。
- 步骤如下：

```
$tar xvfz sqlite-autoconf-3071502.tar.gz
$cd sqlite-autoconf-3071502
$./configure --prefix=/usr/local
$make
$make install
```

上述步骤将在 Linux 机器上安装 SQLite，您可以按照上述讲解的进行验证。

在 Mac OS X 上安装 SQLite

最新版本的 Mac OS X 会预安装 SQLite，但是如果没有可用的安装，只需按照如下步骤进行：

- 请访问 SQLite 下载页面，从源代码区下载 **sqlite-autoconf-*.tar.gz**。
- 步骤如下：

```
$tar xvfz sqlite-autoconf-3071502.tar.gz
$cd sqlite-autoconf-3071502
$./configure --prefix=/usr/local
$make
$make install
```

上述步骤将在 Mac OS X 机器上安装 SQLite，您可以使用下列命令进行验证：

```
$sqlite3
SQLite version 3.7.15.2 2013-01-09 11:53:05
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

最后，在 SQLite 命令提示符下，使用 SQLite 命令做练习。

SQLite 命令

本章将向您讲解 SQLite 编程人员所使用的简单却有用的命令。些命令被称为 SQLite 的点命令，这些命令的不同之处在于它们不以分号（;）结束。

让我们在命令提示符下键入一个简单的 **sqlite3** 命令，在 SQLite 命令提示符下，您可以使用各种 SQLite 命令。

```
$sqlite3
SQLite version 3.3.6
Enter ".help" for instructions
sqlite>
```

如需获取可用的点命令的清单，可以在任何时候输入 ".help"。例如：

```
sqlite>.help
```

上面的命令会显示各种重要的 SQLite 点命令的列表，如下所示：

命令	描述
.backup ? DB? FILE	备份 DB 数据库（默认是 "main"）到 FILE 文件。
.bail ON|OFF	发生错误后停止。默认为 OFF。
.databases	列出附加数据库的名称和文件。
.dump ? TABLE?	以 SQL 文本格式转储数据库。如果指定了 TABLE 表，则只转储匹配 LIKE 模式的 TABLE 表。
.echo ON|OFF	开启或关闭 echo 命令。
.exit	退出 SQLite 提示符。
.explain ON|OFF	开启或关闭适合于 EXPLAIN 的输出模式。如果没有带参数，则为 EXPLAIN on，及开启 EXPLAIN。
.header(s) ON|OFF	开启或关闭头部显示。
.help	显示消息。
.import FILE TABLE	导入来自 FILE 文件的数据到 TABLE 表中。

.indices ?TABLE?	显示所有索引的名称。如果指定了 TABLE 表，则只显示匹配 LIKE 模式的 TABLE 表的索引。
.load FILE ?ENTRY?	加载一个扩展库。
.log FILE|off	开启或关闭日志。FILE 文件可以是 stderr（标准错误）/stdout（标准输出）。
.mode MODE	设置输出模式，MODE 可以是下列之一： **csv** 逗号分隔的值 **column** 左对齐的列 **html** HTML 的 `<table>` 代码 **insert** TABLE 表的 SQL 插入（insert）语句 **line** 每行一个值 **list** 由 .separator 字符串分隔的值 **tabs** 由 Tab 分隔的值 **tcl** TCL 列表元素
.nullvalue STRING	在 NULL 值的地方输出 STRING 字符串。
.output FILENAME	发送输出到 FILENAME 文件。
.output stdout	发送输出到屏幕。
.print STRING...	逐字地输出 STRING 字符串。
.prompt MAIN CONTINUE	替换标准提示符。
.quit	退出 SQLite 提示符。
.read FILENAME	执行 FILENAME 文件中的 SQL。
.schema ?TABLE?	显示 CREATE 语句。如果指定了 TABLE 表，则只显示匹配 LIKE 模式的 TABLE 表。
.separator STRING	改变输出模式和 .import 所使用的分隔符。
.show	显示各种设置的当前值。
.stats ON|OFF	开启或关闭统计。
.tables ?PATTERN?	列出匹配 LIKE 模式的表的名称。
.timeout MS	尝试打开锁定的表 MS 微秒。
.width NUM	

NUM	
.timer ON|OFF	开启或关闭 CPU 定时器测量。

让我们尝试使用 **.show** 命令，来查看 SQLite 命令提示符的默认设置。

```
sqlite>.show
     echo: off
  explain: off
  headers: off
     mode: column
nullvalue: ""
   output: stdout
separator: "|"
    width:
sqlite>
```

> 确保 sqlite> 提示符与点命令之间没有空格，否则将无法正常工作。

格式化输出

您可以使用下列的点命令来格式化输出为本教程下面所列出的格式：

```
sqlite>.header on
sqlite>.mode column
sqlite>.timer on
sqlite>
```

上面设置将产生如下格式的输出：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
CPU Time: user 0.000000 sys 0.000000
```

sqlite_master 表格

主表中保存数据库表的关键信息，并把它命名为 **sqlite_master**。如要查看表概要，可按如下操作：

```
sqlite>.schema sqlite_master
```

这将产生如下结果：

```
CREATE TABLE sqlite_master (
  type text,
  name text,
  tbl_name text,
  rootpage integer,
  sql text
);
```

SQLite 语法

SQLite 是遵循一套独特的称为语法的规则和准则。本教程列出了所有基本的 SQLite 语法，向您提供了一个 SQLite 快速入门。

大小写敏感性

有个重要的点值得注意，SQLite 是不区分大小写的，但也有一些命令是大小写敏感的，比如 **GLOB** 和 **glob** 在 SQLite 的语句中有不同的含义。

注释

SQLite 注释是附加的注释，可以在 SQLite 代码中添加注释以增加其可读性，他们可以出现在任何空白处，包括在表达式内和其他 SQL 语句的中间，但它们不能嵌套。

SQL 注释以两个连续的 "-" 字符（ASCII 0x2d）开始，并扩展至下一个换行符（ASCII 0x0a）或直到输入结束，以先到者为准。

您也可以使用 C 风格的注释，以 "/*" 开始，并扩展至下一个 "*/" 字符对或直到输入结束，以先到者为准。C 庚哥的注释可以跨越多行。

```
sqlite>.help -- This is a single line comment
```

SQLite 语句

所有的 SQLite 语句可以以任何关键字开始，如 SELECT、INSERT、UPDATE、DELETE、ALTER、DROP 等，所有的语句以分号（;）结束。

SQLite ANALYZE 语句：

```
ANALYZE;
or
ANALYZE database_name;
or
ANALYZE database_name.table_name;
```

SQLite AND/OR 子句：

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  CONDITION-1 {AND|OR} CONDITION-2;
```

SQLite ALTER TABLE 语句：

```
ALTER TABLE table_name ADD COLUMN column_def...;
```

SQLite ALTER TABLE 语句（Rename）：

```
ALTER TABLE table_name RENAME TO new_table_name;
```

SQLite ATTACH DATABASE 语句：

```
ATTACH DATABASE 'DatabaseName' As 'Alias-Name';
```

SQLite BEGIN TRANSACTION 语句：

```
BEGIN;
or
BEGIN EXCLUSIVE TRANSACTION;
```

SQLite BETWEEN 子句：

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  column_name BETWEEN val-1 AND val-2;
```

SQLite COMMIT 语句：

```
COMMIT;
```

SQLite CREATE INDEX 语句：

```
CREATE INDEX index_name
ON table_name ( column_name COLLATE NOCASE );
```

SQLite CREATE UNIQUE INDEX 语句：

```
CREATE UNIQUE INDEX index_name
ON table_name ( column1, column2,...columnN);
```

SQLite CREATE TABLE 语句：

```
CREATE TABLE table_name(
   column1 datatype,
   column2 datatype,
   column3 datatype,
   .....
   columnN datatype,
   PRIMARY KEY( one or more columns )
);
```

SQLite CREATE TRIGGER 语句：

```
CREATE TRIGGER database_name.trigger_name
BEFORE INSERT ON table_name FOR EACH ROW
BEGIN
   stmt1;
   stmt2;
   ....
END;
```

SQLite CREATE VIEW 语句：

```
CREATE VIEW database_name.view_name  AS
SELECT statement....;
```

SQLite CREATE VIRTUAL TABLE 语句：

```
CREATE VIRTUAL TABLE database_name.table_name USING weblog( access
or
CREATE VIRTUAL TABLE database_name.table_name USING fts3( );
```

SQLite COMMIT TRANSACTION 语句：

```
COMMIT;
```

SQLite COUNT 子句：

```
SELECT COUNT(column_name)
FROM   table_name
WHERE  CONDITION;
```

SQLite DELETE 语句：

```
DELETE FROM table_name
WHERE  {CONDITION};
```

SQLite DETACH DATABASE 语句：

```
DETACH DATABASE 'Alias-Name';
```

SQLite DISTINCT 子句：

```
SELECT DISTINCT column1, column2....columnN
FROM   table_name;
```

SQLite DROP INDEX 语句：

```
DROP INDEX database_name.index_name;
```

SQLite DROP TABLE 语句：

```
DROP TABLE database_name.table_name;
```

SQLite DROP VIEW 语句：

```
DROP INDEX database_name.view_name;
```

SQLite DROP TRIGGER 语句：

```
DROP INDEX database_name.trigger_name;
```

SQLite EXISTS 子句：

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  column_name EXISTS (SELECT * FROM   table_name );
```

SQLite EXPLAIN 语句：

```
EXPLAIN INSERT statement...;
or
EXPLAIN QUERY PLAN SELECT statement...;
```

SQLite GLOB 子句：

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  column_name GLOB { PATTERN };
```

SQLite GROUP BY 子句：

```
SELECT SUM(column_name)
FROM   table_name
WHERE  CONDITION
GROUP BY column_name;
```

SQLite HAVING 子句：

```
SELECT SUM(column_name)
FROM   table_name
WHERE  CONDITION
GROUP BY column_name
HAVING (arithematic function condition);
```

SQLite INSERT INTO 语句：

```
INSERT INTO table_name( column1, column2....columnN)
VALUES ( value1, value2....valueN);
```

SQLite IN 子句：

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  column_name IN (val-1, val-2,...val-N);
```

SQLite Like 子句：

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  column_name LIKE { PATTERN };
```

SQLite NOT IN 子句：

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  column_name NOT IN (val-1, val-2,...val-N);
```

SQLite ORDER BY 子句：

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  CONDITION
ORDER BY column_name {ASC|DESC};
```

SQLite PRAGMA 语句：

```
PRAGMA pragma_name;

For example:

PRAGMA page_size;
PRAGMA cache_size = 1024;
PRAGMA table_info(table_name);
```

SQLite RELEASE SAVEPOINT 语句：

```
RELEASE savepoint_name;
```

SQLite REINDEX 语句：

```
REINDEX collation_name;
REINDEX database_name.index_name;
REINDEX database_name.table_name;
```

SQLite ROLLBACK 语句：

```
ROLLBACK;
or
ROLLBACK TO SAVEPOINT savepoint_name;
```

SQLite SAVEPOINT 语句：

```
SAVEPOINT savepoint_name;
```

SQLite SELECT 语句：

```
SELECT column1, column2....columnN
FROM   table_name;
```

SQLite UPDATE 语句：

```
UPDATE table_name
SET column1 = value1, column2 = value2....columnN=valueN
[ WHERE  CONDITION ];
```

SQLite VACUUM 语句：

```
VACUUM;
```

SQLite WHERE 子句：

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  CONDITION;
```

SQLite 数据类型

SQLite 数据类型是一个用来指定任何对象的数据类型的属性。SQLite 中的每一列，每个变量和表达式都有相关的数据类型。

您可以在创建表的同时使用这些数据类型。SQLite 使用一个更普遍的动态类型系统。在 SQLite 中，值的数据类型与值本身是相关的，而不是与它的容器相关。

SQLite 存储类

每个存储在 SQLite 数据库中的值都具有以下存储类之一：

存储类	描述
NULL	值是一个 NULL 值。
INTEGER	值是一个带符号的整数，根据值的大小存储在 1、2、3、4、6 或 8 字节中。
REAL	值是一个浮点值，存储为 8 字节的 IEEE 浮点数字。
TEXT	值是一个文本字符串，使用数据库编码（UTF-8、UTF-16BE 或 UTF-16LE）存储。
BLOB	值是一个 blob 数据，完全根据它的输入存储。

SQLite 的存储类稍微比数据类型更普遍。INTEGER 存储类，例如，包含 6 种不同的不同长度的整数数据类型。

SQLite Affinity 类型

SQLite 支持列上的类型 *affinity* 概念。任何列仍然可以存储任何类型的数据，但列的首选存储类是它的 **affinity**。在 SQLite3 数据库中，每个表的列分配为以下类型的 affinity 之一：

Affinity	描述
TEXT	该列使用存储类 NULL、TEXT 或 BLOB 存储所有数据。
NUMERIC	该列可以包含使用所有五个存储类的值。
INTEGER	与带有 NUMERIC affinity 的列相同，在 CAST 表达式中带有异常。
REAL	与带有 NUMERIC affinity 的列相似，不同的是，它会强制把整数值转换为浮点表示。
NONE	带有 affinity NONE 的列，不会优先使用哪个存储类，也不会尝试把数据从一个存储类强制转换为另一个存储类。

SQLite Affinity 及类型名称

下表列出了当创建 SQLite3 表时可使用的各种数据类型名称，同时也显示了相应的应用 Affinity：

数据类型	Affinity
INT INTEGER TINYINT SMALLINT MEDIUMINT BIGINT UNSIGNED BIG INT INT2 INT8	INTEGER
CHARACTER(20) VARCHAR(255) VARYING CHARACTER(255) NCHAR(55) NATIVE CHARACTER(70) NVARCHAR(100) TEXT CLOB	TEXT
BLOB "no datatype specified"	NONE
REAL DOUBLE DOUBLE PRECISION FLOAT	REAL
NUMERIC DECIMAL(10,5) BOOLEAN DATE DATETIME	NUMERIC

Boolean 数据类型

SQLite 没有单独的 Boolean 存储类。相反，布尔值被存储为整数 0（false）和 1（true）。

Date 与 Time 数据类型

SQLite 没有一个单独的用于存储日期和/或时间的存储类，但 SQLite 能够把日期和时间存储为 TEXT、REAL 或 INTEGER 值。

存储类	日期格式
TEXT	格式为 "YYYY-MM-DD HH:MM:SS.SSS" 的日期。
REAL	从公元前 4714 年 11 月 24 日格林尼治时间的正午开始算起的天数。
INTEGER	从 1970-01-01 00:00:00 UTC 算起的秒数。

您可以以任何上述格式来存储日期和时间，并且可以使用内置的日期和时间函数来自由转换不同格式。

SQLite 创建数据库

SQLite 的 **sqlite3** 命令被用来创建新的 SQLite 数据库。您不需要任何特殊的权限即可创建一个数据。

语法

sqlite3 命令的基本语法如下：

```
$sqlite3 DatabaseName.db
```

通常情况下，数据库名称在 RDBMS 内应该是唯一的。

实例

如果您想创建一个新的数据库 <testDB.db>，SQLITE3 语句如下所示：

```
$sqlite3 testDB.db
SQLite version 3.7.15.2 2013-01-09 11:53:05
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

上面的命令将在当前目录下创建一个文件 **testDB.db**。该文件将被 SQLite 引擎用作数据库。如果您已经注意到 sqlite3 命令在成功创建数据库文件之后，将提供一个 **sqlite>** 提示符。

一旦数据库被创建，您就可以使用 SQLite 的 **.databases** 命令来检查它是否在数据库列表中，如下所示：

```
sqlite>.databases
seq  name             file
---  ---------------  ----------------------
0    main             /home/sqlite/testDB.db
```

您可以使用 SQLite **.quit** 命令退出 sqlite 提示符，如下所示：

```
sqlite>.quit
$
```

.dump 命令

您可以在命令提示符中使用 SQLite **.dump** 点命令来导出完整的数据库在一个文本文件中，如下所示：

```
$sqlite3 testDB.db .dump > testDB.sql
```

上面的命令将转换整个 **testDB.db** 数据库的内容到 SQLite 的语句中，并将其转储到 ASCII 文本文件 **testDB.sql** 中。您可以通过简单的方式从生成的 testDB.sql 恢复，如下所示：

```
$sqlite3 testDB.db < testDB.sql
```

此时的数据库是空的，一旦数据库中有表和数据，您可以尝试上述两个程序。现在，让我们继续学习下一章。

SQLite 附加数据库

假设这样一种情况，当在同一时间有多个数据库可用，您想使用其中的任何一个。SQLite 的 **ATTACH DTABASE** 语句是用来选择一个特定的数据库，使用该命令后，所有的 SQLite 语句将在附加的数据库下执行。

语法

SQLite 的 ATTACH DATABASE 语句的基本语法如下：

```
ATTACH DATABASE 'DatabaseName' As 'Alias-Name';
```

如果数据库尚未被创建，上面的命令将创建一个数据库，如果数据库已存在，则把数据库文件名称与逻辑数据库 'Alias-Name' 绑定在一起。

实例

如果想附加一个现有的数据库 **testDB.db**，则 ATTACH DATABASE 语句将如下所示：

```
sqlite> ATTACH DATABASE 'testDB.db' as 'TEST';
```

使用 SQLite **.database** 命令来显示附加的数据库。

```
sqlite> .database
seq  name             file
---  ---------------  ----------------------
0    main             /home/sqlite/testDB.db
2    test             /home/sqlite/testDB.db
```

数据库名称 **main** 和 **temp** 被保留用于主数据库和存储临时表及其他临时数据对象的数据库。这两个数据库名称可用于每个数据库连接，且不应该被用于附加，否则将得到一个警告消息，如下所示：

```
sqlite>  ATTACH DATABASE 'testDB.db' as 'TEMP';
Error: database TEMP is already in use
sqlite>  ATTACH DATABASE 'testDB.db' as 'main';
Error: database TEMP is already in use
```

SQLite 分离数据库

SQLite的 **DETACH DTABASE** 语句是用来把命名数据库从一个数据库连接分离和游离出来，连接是之前使用 ATTACH 语句附加的。如果同一个数据库文件已经被附加上多个别名，DETACH 命令将只断开给定名称的连接，而其余的仍然有效。您无法分离 **main** 或 **temp** 数据库。

> 如果数据库是在内存中或者是临时数据库，则该数据库将被摧毁，且内容将会丢失。

语法

SQLite 的 DETACH DATABASE 'Alias-Name' 语句的基本语法如下：

```
DETACH DATABASE 'Alias-Name';
```

在这里，'Alias-Name' 与您之前使用 ATTACH 语句附加数据库时所用到的别名相同。

实例

假设在前面的章节中您已经创建了一个数据库，并给它附加了 'test' 和 'currentDB'，使用 .database 命令，我们可以看到：

```
sqlite>.databases
seq  name             file
---  ---------------  ----------------------
0    main             /home/sqlite/testDB.db
2    test             /home/sqlite/testDB.db
3    currentDB        /home/sqlite/testDB.db
```

现在，让我们尝试把 'currentDB' 从 testDB.db 中分离出来，如下所示：

```
sqlite> DETACH DATABASE 'currentDB';
```

现在，如果检查当前附加的数据库，您会发现，testDB.db 仍与 'test' 和 'main' 保持连接。

```
sqlite>.databases
seq  name             file
---  ---------------  ----------------------
0    main             /home/sqlite/testDB.db
2    test             /home/sqlite/testDB.db
```

SQLite 创建表

SQLite 的 **CREATE TABLE** 语句用于在任何给定的数据库创建一个新表。创建基本表，涉及到命名表、定义列及每一列的数据类型。

语法

CREATE TABLE 语句的基本语法如下：

```
CREATE TABLE database_name.table_name(
   column1 datatype  PRIMARY KEY(one or more columns),
   column2 datatype,
   column3 datatype,
   .....
   columnN datatype,
);
```

CREATE TABLE 是告诉数据库系统创建一个新表的关键字。CREATE TABLE 语句后跟着表的唯一的名称或标识。您也可以选择指定带有 *table_name* 的 *database_name*。

实例

下面是一个实例，它创建了一个 COMPANY 表，ID 作为主键，NOT NULL 的约束表示在表中创建纪录时这些字段不能为 NULL：

```
sqlite> CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
);
```

让我们再创建一个表，我们将在随后章节的练习中使用：

```
sqlite> CREATE TABLE DEPARTMENT(
   ID INT PRIMARY KEY      NOT NULL,
   DEPT           CHAR(50) NOT NULL,
   EMP_ID         INT      NOT NULL
);
```

您可以使用 SQLIte 命令中的 **.tables** 命令来验证表是否已成功创建，该命令用于列出附加数据库中的所有表。

```
sqlite>.tables
COMPANY     DEPARTMENT
```

在这里，可以看到 COMPANY 表出现两次，一个是主数据库的 COMPANY 表，一个是为 testDB.db 创建的 'test' 别名的 test.COMPANY 表。您可以使用 SQLite **.schema** 命令得到表的完整信息，如下所示：

```
sqlite>.schema COMPANY
CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
);
```

SQLite 删除表

SQLite 的 **DROP TABLE** 语句用来删除表定义及其所有相关数据、索引、触发器、约束和该表的权限规范。

> 使用此命令时要特别注意，因为一旦一个表被删除，表中所有信息也将永远丢失。

语法

DROP TABLE 语句的基本语法如下。您可以选择指定带有表名的数据库名称，如下所示：

```
DROP TABLE database_name.table_name;
```

实例

让我们先确认 COMPANY 表已经存在，然后我们将其从数据库中删除。

```
sqlite>.tables
COMPANY         test.COMPANY
```

这意味着 COMPANY 表已存在数据库中，接下来让我们把它从数据库中删除，如下：

```
sqlite>DROP TABLE COMPANY;
sqlite>
```

现在，如果尝试 .TABLES 命令，那么将无法找到 COMPANY 表了：

```
sqlite>.tables
sqlite>
```

显示结果为空，意味着已经成功从数据库删除表。

SQLite Insert 语句

SQLite 的 **INSERT INTO** 语句用于向数据库的某个表中添加新的数据行。

语法

INSERT INTO 语句有两种基本语法，如下所示：

```
INSERT INTO TABLE_NAME (column1, column2, column3,...columnN)]
VALUES (value1, value2, value3,...valueN);
```

在这里，column1, column2,...columnN 是要插入数据的表中的列的名称。

如果要为表中的所有列添加值，您也可以不需要在 SQLite 查询中指定列名称。但要确保值的顺序与列在表中的顺序一致。SQLite 的 INSERT INTO 语法如下：

```
INSERT INTO TABLE_NAME VALUES (value1,value2,value3,...valueN);
```

实例

假设您已经在 testDB.db 中创建了 COMPANY表，如下所示：

```
sqlite> CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
);
```

现在，下面的语句将在 COMPANY 表中创建六个记录：

```
INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Paul', 32, 'California', 20000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (2, 'Allen', 25, 'Texas', 15000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (3, 'Teddy', 23, 'Norway', 20000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (5, 'David', 27, 'Texas', 85000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (6, 'Kim', 22, 'South-Hall', 45000.00 );
```

您也可以使用第二种语法在 COMPANY 表中创建一个记录，如下所示：

```
INSERT INTO COMPANY VALUES (7, 'James', 24, 'Houston', 10000.00 );
```

上面的所有语句将在 COMPANY 表中创建下列记录。下一章会教您如何从一个表中显示所有这些记录。

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

使用一个表来填充另一个表

您可以通过在一个有一组字段的表上使用 select 语句，填充数据到零一个表中。下面是语法：

```
INSERT INTO first_table_name [(column1, column2, ... columnN)]
   SELECT column1, column2, ...columnN
   FROM second_table_name
   [WHERE condition];
```

您暂时可以先跳过上面的语句，可以先学习后面章节中介绍的 SELECT 和 WHERE 子句。

SQLite Select 语句

SQLite 的 **SELECT** 语句用于从 SQLite 数据库表中获取数据，以结果表的形式返回数据。这些结果表也被称为结果集。

语法

SQLite 的 SELECT 语句的基本语法如下：

```
SELECT column1, column2, columnN FROM table_name;
```

在这里，column1, column2...是表的字段，他们的值即是您要获取的。如果您想获取所有可用的字段，那么可以使用下面的语法：

```
SELECT * FROM table_name;
```

实例

假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面是一个实例，使用 SELECT 语句获取并显示所有这些记录。在这里，前三个命令被用来设置正确格式化的输出。

```
sqlite>.header on
sqlite>.mode column
sqlite> SELECT * FROM COMPANY;
```

最后，将得到以下的结果：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

如果只想获取 COMPANY 表中指定的字段，则使用下面的查询：

```
sqlite> SELECT ID, NAME, SALARY FROM COMPANY;
```

上面的查询会产生以下结果：

```
ID          NAME        SALARY
----------  ----------  ----------
1           Paul        20000.0
2           Allen       15000.0
3           Teddy       20000.0
4           Mark        65000.0
5           David       85000.0
6           Kim         45000.0
7           James       10000.0
```

设置输出列的宽度

有时，由于要显示的列的默认宽度导致 **.mode column**，这种情况下，输出被截断。此时，您可以使用 **.width num, num....** 命令设置显示列的宽度，如下所示：

```
sqlite>.width 10, 20, 10
sqlite>SELECT * FROM COMPANY;
```

上面的 **.width** 命令设置第一列的宽度为 10，第二列的宽度为 20，第三列的宽度为 10。因此上述 SELECT 语句将得到以下结果：

```
ID          NAME                  AGE         ADDRESS     SALARY
----------  --------------------  ----------  ----------  ---------
1           Paul                  32          California  20000.0
2           Allen                 25          Texas       15000.0
3           Teddy                 23          Norway      20000.0
4           Mark                  25          Rich-Mond   65000.0
5           David                 27          Texas       85000.0
6           Kim                   22          South-Hall  45000.0
7           James                 24          Houston     10000.0
```

Schema 信息

因为所有的点命令只在 SQLite 提示符中可用，所以当您进行带有 SQLite 的编程时，您要使用下面的带有 **sqlite_master** 表的 SELECT 语句来列出所有在数据库中创建的表：

```
sqlite> SELECT tbl_name FROM sqlite_master WHERE type = 'table';
```

假设在 testDB.db 中已经存在唯一的 COMPANY 表，则将产生以下结果：

```
tbl_name
----------
COMPANY
```

您可以列出关于 COMPANY 表的完整信息，如下所示：

```
sqlite> SELECT sql FROM sqlite_master WHERE type = 'table' AND tbl_
```

假设在 testDB.db 中已经存在唯一的 COMPANY 表，则将产生以下结果：

```
CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
)
```

SQLite 运算符

SQLite 运算符是什么？

运算符是一个保留字或字符，主要用于 SQLite 语句的 WHERE 子句中执行操作，如比较和算术运算。

运算符用于指定 SQLite 语句中的条件，并在语句中连接多个条件。

- 算术运算符
- 比较运算符
- 逻辑运算符
- 位运算符

SQLite 算术运算符

假设变量 a=10，变量 b=20，则：

运算符	描述	实例
+	加法 - 把运算符两边的值相加	a + b 将得到 30
-	减法 - 左操作数减去右操作数	a - b 将得到 -10
*	乘法 - 把运算符两边的值相乘	a * b 将得到 200
/	除法 - 左操作数除以右操作数	b / a 将得到 2
%	取模 - 左操作数除以右操作数后得到的余数	b % a will give 0

实例

下面是 SQLite 算术运算符的简单实例：

```
sqlite> .mode line
sqlite> select 10 + 20;
10 + 20 = 30

sqlite> select 10 - 20;
10 - 20 = -10

sqlite> select 10 * 20;
10 * 20 = 200

sqlite> select 10 / 5;
10 / 5 = 2

sqlite> select 12 %  5;
12 %  5 = 2
```

SQLite 比较运算符

假设变量 a=10，变量 b=20，则：

运算符	描述	实例
==	检查两个操作数的值是否相等，如果相等则条件为真。	(a == b) 不为真。
=	检查两个操作数的值是否相等，如果相等则条件为真。	(a = b) 不为真。
!=	检查两个操作数的值是否相等，如果不相等则条件为真。	(a != b) 为真。
<>	检查两个操作数的值是否相等，如果不相等则条件为真。	(a <> b) 为真。
>	检查左操作数的值是否大于右操作数的值，如果是则条件为真。	(a > b) 不为真。
<	检查左操作数的值是否小于右操作数的值，如果是则条件为真。	(a < b) 为真。
>=	检查左操作数的值是否大于等于右操作数的值，如果是则条件为真。	(a >= b) 不为真。
<=	检查左操作数的值是否小于等于右操作数的值，如果是则条件为真。	(a <= b) 为真。
!<	检查左操作数的值是否不小于右操作数的值，如果是则条件为真。	(a !< b) 为假。
!>	检查左操作数的值是否不大于右操作数的值，如果是则条件为真。	(a !> b) 为真。

实例

假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面的实例演示了各种 SQLite 比较运算符的用法。

> 在这里，我们使用 **WHERE** 子句，这将会在后边单独的一个章节中讲解，但现在您需要明白，WHERE 子句是用来设置 SELECT 语句的条件语句。

下面的 SELECT 语句列出了 SALARY 大于 50,000.00 的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE SALARY > 50000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

下面的 SELECT 语句列出了 SALARY 等于 20,000.00 的所有记录：

```
sqlite>  SELECT * FROM COMPANY WHERE SALARY = 20000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
3           Teddy       23          Norway      20000.0
```

下面的 SELECT 语句列出了 SALARY 不等于 20,000.00 的所有记录：

```
sqlite>  SELECT * FROM COMPANY WHERE SALARY != 20000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面的 SELECT 语句列出了 SALARY 不等于 20,000.00 的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE SALARY <> 20000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面的 SELECT 语句列出了 SALARY 大于等于 65,000.00 的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE SALARY >= 65000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

SQLite 逻辑运算符

下面是 SQLite 中所有的逻辑运算符列表。

运算符	描述
AND	AND 运算符允许在一个 SQL 语句的 WHERE 子句中的多个条件的存在。
BETWEEN	BETWEEN 运算符用于在给定最小值和最大值范围内的一系列值中搜索值。
EXISTS	EXISTS 运算符用于在满足一定条件的指定表中搜索行的存在。
IN	IN 运算符用于把某个值与一系列指定列表的值进行比较。
NOT IN	IN 运算符的对立面，用于把某个值与不在一系列指定列表的值进行比较。
LIKE	LIKE 运算符用于把某个值与使用通配符运算符的相似值进行比较。
GLOB	GLOB 运算符用于把某个值与使用通配符运算符的相似值进行比较。GLOB 与 LIKE 不同之处在于，它是大小写敏感的。
NOT	NOT 运算符是所用的逻辑运算符的对立面。比如 NOT EXISTS、NOT BETWEEN、NOT IN，等等。它是否定运算符。
OR	OR 运算符用于结合一个 SQL 语句的 WHERE 子句中的多个条件。
IS NULL	NULL 运算符用于把某个值与 NULL 值进行比较。
IS	IS 运算符与 = 相似。
IS NOT	IS NOT 运算符与 != 相似。
||	连接两个不同的字符串，得到一个新的字符串。
UNIQUE	UNIQUE 运算符搜索指定表中的每一行，确保唯一性（无重复）。

实例

假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面的实例演示了 SQLite 逻辑运算符的用法。

下面的 SELECT 语句列出了 AGE 大于等于 25 且工资大于等于 65000.00 的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE AGE >= 25 AND SALARY >= 65000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

下面的 SELECT 语句列出了 AGE 大于等于 25 或工资大于等于 65000.00 的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE AGE >= 25 OR SALARY >= 65000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

下面的 SELECT 语句列出了 AGE 不为 NULL 的所有记录，结果显示所有的记录，意味着没有一个记录的 AGE 等于 NULL：

```
sqlite>  SELECT * FROM COMPANY WHERE AGE IS NOT NULL;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面的 SELECT 语句列出了 NAME 以 'Ki' 开始的所有记录，'Ki' 之后的字符不做限制：

```
sqlite> SELECT * FROM COMPANY WHERE NAME LIKE 'Ki%';
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
6           Kim         22          South-Hall  45000.0
```

下面的 SELECT 语句列出了 NAME 以 'Ki' 开始的所有记录，'Ki' 之后的字符不做限制：

```
sqlite> SELECT * FROM COMPANY WHERE NAME GLOB 'Ki*';
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
6           Kim         22          South-Hall  45000.0
```

下面的 SELECT 语句列出了 AGE 的值为 25 或 27 的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE AGE IN ( 25, 27 );
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

下面的 SELECT 语句列出了 AGE 的值既不是 25 也不是 27 的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE AGE NOT IN ( 25, 27 );
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
3           Teddy       23          Norway      20000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面的 SELECT 语句列出了 AGE 的值在 25 与 27 之间的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE AGE BETWEEN 25 AND 27;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

下面的 SELECT 语句使用 SQL 子查询，子查询查找 SALARY > 65000 的带有 AGE 字段的所有记录，后边的 WHERE 子句与 EXISTS 运算符一起使用，列出了外查询中的 AGE 存在于子查询返回的结果中的所有记录：

```
sqlite> SELECT AGE FROM COMPANY
        WHERE EXISTS (SELECT AGE FROM COMPANY WHERE SALARY > 65000)
AGE
----------
32
25
23
25
27
22
24
```

下面的 SELECT 语句使用 SQL 子查询，子查询查找 SALARY > 65000 的带有 AGE 字段的所有记录，后边的 WHERE 子句与 > 运算符一起使用，列出了外查询中的 AGE 大于子查询返回的结果中的年龄的所有记录：

```
sqlite> SELECT * FROM COMPANY
        WHERE AGE > (SELECT AGE FROM COMPANY WHERE SALARY > 65000);
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
```

SQLite 位运算符

位运算符作用于位，并逐位执行操作。真值表 & 和 | 如下：

<table>
<tr><th>p</th><th>q</th><th>p & q</th><th>p | q</th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>1</td></tr>
</table>

假设如果 A = 60，且 B = 13，现在以二进制格式，它们如下所示：

```
A = 0011 1100

B = 0000 1101

-----------------

A&B = 0000 1100

A|B = 0011 1101

~A  = 1100 0011
```

下表中列出了 SQLite 语言支持的位运算符。假设变量 A=60，变量 B=13，则：

<table>
<tr><th>运算符</th><th>描述</th><th>实例</th></tr>
<tr><td>&</td><td>如果同时存在于两个操作数中，二进制 AND 运算符复制一位到结果中。</td><td>(A & B) 将得到 12，即为 0000 1100</td></tr>
<tr><td>|</td><td>如果存在于任一操作数中，二进制 OR 运算符复制一位到结果中。</td><td>(A | B) 将得到 61，即为 0011 1101</td></tr>
<tr><td>~</td><td>二进制补码运算符是一元运算符，具有"翻转"位效应。</td><td>(~A) 将得到 -61，即为 1100 0011，2 的补码形式，带符号的二进制数。</td></tr>
<tr><td><<</td><td>二进制左移运算符。左操作数的值向左移动右操作数指定的位数。</td><td>A << 2 将得到 240，即为 1111 0000</td></tr>
<tr><td>>></td><td>二进制右移运算符。左操作数的值向右移动右操作数指定的位数。</td><td>A >> 2 将得到 15，即为 0000 1111</td></tr>
</table>

实例

下面的实例演示了 SQLite 位运算符的用法：

```
sqlite> .mode line
sqlite> select 60 | 13;
60 | 13 = 61

sqlite> select 60 & 13;
60 & 13 = 12

sqlite> select  60 ^ 13;
10 * 20 = 200

sqlite>  select  (~60);
(~60) = -61

sqlite>  select  (60 << 2);
(60 << 2) = 240

sqlite>  select  (60 >> 2);
(60 >> 2) = 15
```

SQLite 表达式

表达式是一个或多个值、运算符和计算值的SQL函数的组合。

SQL 表达式与公式类似，都写在查询语言中。您还可以使用特定的数据集来查询数据库。

语法

假设 SELECT 语句的基本语法如下：

```
SELECT column1, column2, columnN
FROM table_name
WHERE [CONTION | EXPRESSION];
```

有不同类型的 SQLite 表达式，具体讲解如下：

SQLite - 布尔表达式

SQLite 的布尔表达式在匹配单个值的基础上获取数据。语法如下：

```
SELECT column1, column2, columnN
FROM table_name
WHERE SINGLE VALUE MATCHTING EXPRESSION;
```

假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面的实例演示了 SQLite 布尔表达式的用法：

```
sqlite> SELECT * FROM COMPANY WHERE SALARY = 10000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
4           James        24          Houston   10000.0
```

SQLite - 数值表达式

这些表达式用来执行查询中的任何数学运算。语法如下：

```
SELECT numerical_expression as  OPERATION_NAME
[FROM table_name WHERE CONDITION] ;
```

在这里，numerical_expression 用于数学表达式或任何公式。下面的实例演示了 SQLite 数值表达式的用法：

```
sqlite> SELECT (15 + 6) AS ADDITION
ADDITION = 21
```

有几个内置的函数，比如 avg()、sum()、count()，等等，执行被称为对一个表或一个特定的表列的汇总数据计算。

```
sqlite> SELECT COUNT(*) AS "RECORDS" FROM COMPANY;
RECORDS = 7
```

SQLite - 日期表达式

日期表达式返回当前系统日期和时间值，这些表达式将被用于各种数据操作。

```
sqlite>  SELECT CURRENT_TIMESTAMP;
CURRENT_TIMESTAMP = 2013-03-17 10:43:35
```

SQLite Where 子句

SQLite的 **WHERE** 子句用于指定从一个表或多个表中获取数据的条件。

如果满足给定的条件，即为真（true）时，则从表中返回特定的值。您可以使用 WHERE 子句来过滤记录，只获取需要的记录。

WHERE 子句不仅可用在 SELECT 语句中，它也可用在 UPDATE、DELETE 语句中，等等，这些我们将在随后的章节中学习到。

语法

SQLite 的带有 WHERE 子句的 SELECT 语句的基本语法如下：

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition]
```

实例

您还可以使用比较或逻辑运算符指定条件，比如 >、<、=、LIKE、NOT，等等。假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面的实例演示了 SQLite 逻辑运算符的用法。下面的 SELECT 语句列出了 AGE 大于等于 25 且工资大于等于 65000.00 的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE AGE >= 25 AND SALARY >= 65000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

下面的 SELECT 语句列出了 AGE 大于等于 25 或工资大于等于 65000.00 的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE AGE >= 25 OR SALARY >= 65000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

下面的 SELECT 语句列出了 AGE 不为 NULL 的所有记录，结果显示所有的记录，意味着没有一个记录的 AGE 等于 NULL：

```
sqlite>  SELECT * FROM COMPANY WHERE AGE IS NOT NULL;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面的 SELECT 语句列出了 NAME 以 'Ki' 开始的所有记录，'Ki' 之后的字符不做限制：

```
sqlite> SELECT * FROM COMPANY WHERE NAME LIKE 'Ki%';
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
6           Kim         22          South-Hall  45000.0
```

下面的 SELECT 语句列出了 NAME 以 'Ki' 开始的所有记录，'Ki' 之后的字符不做限制：

```
sqlite> SELECT * FROM COMPANY WHERE NAME GLOB 'Ki*';
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
6           Kim         22          South-Hall  45000.0
```

下面的 SELECT 语句列出了 AGE 的值为 25 或 27 的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE AGE IN ( 25, 27 );
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

下面的 SELECT 语句列出了 AGE 的值既不是 25 也不是 27 的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE AGE NOT IN ( 25, 27 );
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
3           Teddy       23          Norway      20000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面的 SELECT 语句列出了 AGE 的值在 25 与 27 之间的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE AGE BETWEEN 25 AND 27;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

下面的 SELECT 语句使用 SQL 子查询，子查询查找 SALARY > 65000 的带有 AGE 字段的所有记录，后边的 WHERE 子句与 EXISTS 运算符一起使用，列出了外查询中的 AGE 存在于子查询返回的结果中的所有记录：

```
sqlite> SELECT AGE FROM COMPANY
        WHERE EXISTS (SELECT AGE FROM COMPANY WHERE SALARY > 65000)
AGE
----------
32
25
23
25
27
22
24
```

下面的 SELECT 语句使用 SQL 子查询，子查询查找 SALARY > 65000 的带有 AGE 字段的所有记录，后边的 WHERE 子句与 > 运算符一起使用，列出了外查询中的 AGE 大于子查询返回的结果中的年龄的所有记录：

```
sqlite> SELECT * FROM COMPANY
        WHERE AGE > (SELECT AGE FROM COMPANY WHERE SALARY > 65000);
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
```

SQLite AND/OR 运算符

SQLite 的 **AND** 和 **OR** 运算符用于编译多个条件来缩小在 SQLite 语句中所选的数据。这两个运算符被称为连接运算符。

这些运算符为同一个 SQLite 语句中不同的运算符之间的多个比较提供了可能。

AND 运算符

AND 运算符允许在一个 SQL 语句的 WHERE 子句中的多个条件的存在。使用 AND 运算符时，只有当所有条件都为真（true）时，整个条件为真（true）。例如，只有当 condition1 和 condition2 都为真（true）时，[condition1] AND [condition2] 为真（true）。

语法

带有 WHERE 子句的 AND 运算符的基本语法如下：

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition1] AND [condition2]...AND [conditionN];
```

您可以使用 AND 运算符来结合 N 个数量的条件。SQLite 语句需要执行的动作是，无论是事务或查询，所有由 AND 分隔的条件都必须为真（TRUE）。

实例

假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面的 SELECT 语句列出了 AGE 大于等于 25 且工资大于等于 65000.00 的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE AGE >= 25 AND SALARY >= 65000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

OR 运算符

OR 运算符也用于结合一个 SQL 语句的 WHERE 子句中的多个条件。使用 OR 运算符时，只要当条件中任何一个为真（true）时，整个条件为真（true）。例如，只要当 condition1 或 condition2 有一个为真（true）时，[condition1] OR [condition2] 为真（true）。

语法

带有 WHERE 子句的 OR 运算符的基本语法如下：

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition1] OR [condition2]...OR [conditionN]
```

您可以使用 OR 运算符来结合 N 个数量的条件。SQLite 语句需要执行的动作是，无论是事务或查询，只要任何一个由 OR 分隔的条件为真（TRUE）即可。

实例

假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面的 SELECT 语句列出了 AGE 大于等于 25 或工资大于等于 65000.00 的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE AGE >= 25 OR SALARY >= 65000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

SQLite Update 语句

SQLite 的 **UPDATE** 查询用于修改表中已有的记录。可以使用带有 WHERE 子句的 UPDATE 查询来更新选定行，否则所有的行都会被更新。

语法

带有 WHERE 子句的 UPDATE 查询的基本语法如下：

```
UPDATE table_name
SET column1 = value1, column2 = value2...., columnN = valueN
WHERE [condition];
```

您可以使用 AND 或 OR 运算符来结合 N 个数量的条件。

实例

假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面是一个实例，它会更新 ID 为 6 的客户地址：

```
sqlite> UPDATE COMPANY SET ADDRESS = 'Texas' WHERE ID = 6;
```

现在，COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          Texas       45000.0
7           James       24          Houston     10000.0
```

如果您想修改 COMPANY 表中 ADDRESS 和 SALARY 列的所有值，则不需要使用 WHERE 子句，UPDATE 查询如下：

```
sqlite> UPDATE COMPANY SET ADDRESS = 'Texas', SALARY = 20000.00;
```

现在，COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          Texas       20000.0
2           Allen       25          Texas       20000.0
3           Teddy       23          Texas       20000.0
4           Mark        25          Texas       20000.0
5           David       27          Texas       20000.0
6           Kim         22          Texas       20000.0
7           James       24          Texas       20000.0
```

SQLite Delete 语句

SQLite 的 **DELETE** 查询用于删除表中已有的记录。可以使用带有 WHERE 子句的 DELETE 查询来删除选定行，否则所有的记录都会被删除。

语法

带有 WHERE 子句的 DELETE 查询的基本语法如下：

```
DELETE FROM table_name
WHERE [condition];
```

您可以使用 AND 或 OR 运算符来结合 N 个数量的条件。

实例

假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面是一个实例，它会删除 ID 为 7 的客户：

```
sqlite> DELETE FROM COMPANY WHERE ID = 7;
```

现在，COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
```

如果您想要从 COMPANY 表中删除所有记录，则不需要使用 WHERE 子句，DELETE 查询如下：

```
sqlite> DELETE FROM COMPANY;
```

现在，COMPANY 表中没有任何的记录，因为所有的记录已经通过 DELETE 语句删除。

SQLite Like 子句

SQLite 的 **LIKE** 运算符是用来匹配通配符指定模式的文本值。如果搜索表达式与模式表达式匹配，LIKE 运算符将返回真（true），也就是 1。这里有两个通配符与 LIKE 运算符一起使用：

- 百分号 （%）
- 下划线 （_）

百分号（%）代表零个、一个或多个数字或字符。下划线（_）代表一个单一的数字或字符。这些符号可以被组合使用。

语法

% 和 _ 的基本语法如下：

```
SELECT FROM table_name
WHERE column LIKE 'XXXX%'

or

SELECT FROM table_name
WHERE column LIKE '%XXXX%'

or

SELECT FROM table_name
WHERE column LIKE 'XXXX_'

or

SELECT FROM table_name
WHERE column LIKE '_XXXX'

or

SELECT FROM table_name
WHERE column LIKE '_XXXX_'
```

您可以使用 AND 或 OR 运算符来结合 N 个数量的条件。在这里，XXXX 可以是任何数字或字符串值。

实例

下面一些实例演示了 带有 '%' 和 '_' 运算符的 LIKE 子句不同的地方：

语句	描述
WHERE SALARY LIKE '200%'	查找以 200 开头的任意值
WHERE SALARY LIKE '%200%'	查找任意位置包含 200 的任意值
WHERE SALARY LIKE '_00%'	查找第二位和第三位为 00 的任意值
WHERE SALARY LIKE '2_%_%'	查找以 2 开头，且长度至少为 3 个字符的任意值
WHERE SALARY LIKE '%2'	查找以 2 结尾的任意值
WHERE SALARY LIKE '_2%3'	查找第二位为 2，且以 3 结尾的任意值
WHERE SALARY LIKE '2___3'	查找长度为 5 位数，且以 2 开头以 3 结尾的任意值

让我们举一个实际的例子，假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面是一个实例，它显示 COMPANY 表中 AGE 以 2 开头的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE AGE  LIKE '2%';
```

这将产生以下结果：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面是一个实例，它显示 COMPANY 表中 ADDRESS 文本里包含一个连字符（-）的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE ADDRESS  LIKE '%-%';
```

这将产生以下结果：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
4           Mark        25          Rich-Mond   65000.0
6           Kim         22          South-Hall  45000.0
```

SQLite Glob 子句

SQLite 的 **GLOB** 运算符是用来匹配通配符指定模式的文本值。如果搜索表达式与模式表达式匹配，GLOB 运算符将返回真（true），也就是 1。与 LIKE 运算符不同的是，GLOB 是大小写敏感的，对于下面的通配符，它遵循 UNIX 的语法。

- 星号 （*）
- 问号 （?）

星号（*）代表零个、一个或多个数字或字符。问号（?）代表一个单一的数字或字符。这些符号可以被组合使用。

语法

- 和 ? 的基本语法如下：

```
SELECT FROM table_name
WHERE column GLOB 'XXXX*'

or

SELECT FROM table_name
WHERE column GLOB '*XXXX*'

or

SELECT FROM table_name
WHERE column GLOB 'XXXX?'

or

SELECT FROM table_name
WHERE column GLOB '?XXXX'

or

SELECT FROM table_name
WHERE column GLOB '?XXXX?'

or

SELECT FROM table_name
WHERE column GLOB '????'
```

您可以使用 AND 或 OR 运算符来结合 N 个数量的条件。在这里，XXXX 可以是任何数字或字符串值。

实例

下面一些实例演示了 带有 '*' 和 '?' 运算符的 GLOB 子句不同的地方：

语句	描述
WHERE SALARY GLOB '200*'	查找以 200 开头的任意值
WHERE SALARY GLOB '*200*'	查找任意位置包含 200 的任意值
WHERE SALARY GLOB '?00*'	查找第二位和第三位为 00 的任意值
WHERE SALARY GLOB '2??'	查找以 2 开头，且长度至少为 3 个字符的任意值
WHERE SALARY GLOB '*2'	查找以 2 结尾的任意值
WHERE SALARY GLOB '?2*3'	查找第二位为 2，且以 3 结尾的任意值
WHERE SALARY GLOB '2???3'	查找长度为 5 位数，且以 2 开头以 3 结尾的任意值

让我们举一个实际的例子，假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面是一个实例，它显示 COMPANY 表中 AGE 以 2 开头的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE AGE  GLOB '2*';
```

这将产生以下结果：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面是一个实例，它显示 COMPANY 表中 ADDRESS 文本里包含一个连字符（-）的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE ADDRESS  GLOB '*-*';
```

这将产生以下结果：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
4           Mark        25          Rich-Mond   65000.0
6           Kim         22          South-Hall  45000.0
```

SQLite Limit 子句

SQLite 的 **LIMIT** 子句用于限制由 SELECT 语句返回的数据数量。

语法

带有 LIMIT 子句的 SELECT 语句的基本语法如下：

```
SELECT column1, column2, columnN
FROM table_name
LIMIT [no of rows]
```

下面是 LIMIT 子句与 OFFSET 子句一起使用时的语法：

```
SELECT column1, column2, columnN
FROM table_name
LIMIT [no of rows] OFFSET [row num]
```

SQLite 引擎将返回从下一行开始直到给定的 OFFSET 为止的所有行，如下面的最后一个实例所示。

实例

假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面是一个实例，它限制了您想要从表中提取的行数：

```
sqlite> SELECT * FROM COMPANY LIMIT 6;
```

这将产生以下结果：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
```

但是，在某些情况下，可能需要从一个特定的偏移开始提取记录。下面是一个实例，从第三位开始提取 3 个记录：

```
sqlite> SELECT * FROM COMPANY LIMIT 3 OFFSET 2;
```

这将产生以下结果：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

SQLite Order By

SQLite 的 **ORDER BY** 子句是用来基于一个或多个列按升序或降序顺序排列数据。

语法

ORDER BY 子句的基本语法如下：

```
SELECT column-list
FROM table_name
[WHERE condition]
[ORDER BY column1, column2, .. columnN] [ASC | DESC];
```

您可以在 ORDER BY 子句中使用多个列。确保您使用的排序列在列清单中。

实例

假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面是一个实例，它会将结果按 SALARY 降序排序：

```
sqlite> SELECT * FROM COMPANY ORDER BY SALARY ASC;
```

这将产生以下结果：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
7           James       24          Houston     10000.0
2           Allen       25          Texas       15000.0
1           Paul        32          California  20000.0
3           Teddy       23          Norway      20000.0
6           Kim         22          South-Hall  45000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

下面是一个实例，它会将结果按 NAME 和 SALARY 降序排序：

```
sqlite> SELECT * FROM COMPANY ORDER BY NAME, SALARY ASC;
```

这将产生以下结果：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000.0
5           David       27          Texas       85000.0
7           James       24          Houston     10000.0
6           Kim         22          South-Hall  45000.0
4           Mark        25          Rich-Mond   65000.0
1           Paul        32          California  20000.0
3           Teddy       23          Norway      20000.0
```

下面是一个实例，它会将结果按 NAME 降序排序：

```
sqlite> SELECT * FROM COMPANY ORDER BY NAME DESC;
```

这将产生以下结果：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
3           Teddy       23          Norway      20000.0
1           Paul        32          California  20000.0
4           Mark        25          Rich-Mond   65000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
5           David       27          Texas       85000.0
2           Allen       25          Texas       15000.0
```

SQLite Group By

SQLite 的 **GROUP BY** 子句用于与 SELECT 语句一起使用，来对相同的数据进行分组。

在 SELECT 语句中，GROUP BY 子句放在 WHERE 子句之后，放在 ORDER BY 子句之前。

语法

下面给出了 GROUP BY 子句的基本语法。GROUP BY 子句必须放在 WHERE 子句中的条件之后，必须放在 ORDER BY 子句之前。

```
SELECT column-list
FROM table_name
WHERE [ conditions ]
GROUP BY column1, column2....columnN
ORDER BY column1, column2....columnN
```

您可以在 GROUP BY 子句中使用多个列。确保您使用的分组列在列清单中。

实例

假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

如果您想了解每个客户的工资总额，则可使用 GROUP BY 查询，如下所示：

```
sqlite> SELECT NAME, SUM(SALARY) FROM COMPANY GROUP BY NAME;
```

这将产生以下结果：

```
NAME        SUM(SALARY)
----------  -----------
Allen       15000.0
David       85000.0
James       10000.0
Kim         45000.0
Mark        65000.0
Paul        20000.0
Teddy       20000.0
```

现在，让我们使用下面的 INSERT 语句在 COMPANY 表中另外创建三个记录：

```
INSERT INTO COMPANY VALUES (8, 'Paul', 24, 'Houston', 20000.00 );
INSERT INTO COMPANY VALUES (9, 'James', 44, 'Norway', 5000.00 );
INSERT INTO COMPANY VALUES (10, 'James', 45, 'Texas', 5000.00 );
```

现在，我们的表具有重复名称的记录，如下所示：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
8           Paul        24          Houston     20000.0
9           James       44          Norway      5000.0
10          James       45          Texas       5000.0
```

让我们用同样的 GROUP BY 语句来对所有记录按 NAME 列进行分组，如下所示：

```
sqlite> SELECT NAME, SUM(SALARY) FROM COMPANY GROUP BY NAME ORDER E
```

这将产生以下结果：

```
NAME        SUM(SALARY)
----------  -----------
Allen       15000
David       85000
James       20000
Kim         45000
Mark        65000
Paul        40000
Teddy       20000
```

让我们把 ORDER BY 子句与 GROUP BY 子句一起使用，如下所示：

```
sqlite>  SELECT NAME, SUM(SALARY)
         FROM COMPANY GROUP BY NAME ORDER BY NAME DESC;
```

这将产生以下结果：

```
NAME        SUM(SALARY)
----------  -----------
Teddy       20000
Paul        40000
Mark        65000
Kim         45000
James       20000
David       85000
Allen       15000
```

SQLite Having 子句

HAVING 子句允许指定条件来过滤将出现在最终结果中的分组结果。

WHERE 子句在所选列上设置条件，而 HAVING 子句则在由 GROUP BY 子句创建的分组上设置条件。

语法

下面是 HAVING 子句在 SELECT 查询中的位置：

```
SELECT
FROM
WHERE
GROUP BY
HAVING
ORDER BY
```

在一个查询中，HAVING 子句必须放在 GROUP BY 子句之后，必须放在 ORDER BY 子句之前。下面是包含 HAVING 子句的 SELECT 语句的语法：

```
SELECT column1, column2
FROM table1, table2
WHERE [ conditions ]
GROUP BY column1, column2
HAVING [ conditions ]
ORDER BY column1, column2
```

实例

假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
8           Paul        24          Houston     20000.0
9           James       44          Norway      5000.0
10          James       45          Texas       5000.0
```

下面是一个实例，它将显示名称计数小于 2 的所有记录：

```
sqlite > SELECT * FROM COMPANY GROUP BY name HAVING count(name) <
```

这将产生以下结果：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000
5           David       27          Texas       85000
6           Kim         22          South-Hall  45000
4           Mark        25          Rich-Mond   65000
3           Teddy       23          Norway      20000
```

下面是一个实例，它将显示名称计数大于 2 的所有记录：

```
sqlite > SELECT * FROM COMPANY GROUP BY name HAVING count(name) >
```

这将产生以下结果：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
10          James       45          Texas       5000
```

SQLite Distinct 关键字

SQLite 的 **DISTINCT** 关键字与 SELECT 语句一起使用，来消除所有重复的记录，并只获取唯一一次记录。

有可能出现一种情况，在一个表中有多个重复的记录。当提取这样的记录时，DISTINCT 关键字就显得特别有意义，它只获取唯一一次记录，而不是获取重复记录。

语法

用于消除重复记录的 DISTINCT 关键字的基本语法如下：

```
SELECT DISTINCT column1, column2,.....columnN
FROM table_name
WHERE [condition]
```

实例

假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
8           Paul        24          Houston     20000.0
9           James       44          Norway      5000.0
10          James       45          Texas       5000.0
```

首先，让我们来看看下面的 SELECT 查询，它将返回重复的工资记录：

```
sqlite> SELECT name FROM COMPANY;
```

这将产生以下结果：

```
NAME
----------
Paul
Allen
Teddy
Mark
David
Kim
James
Paul
James
James
```

现在，让我们在上述的 SELECT 查询中使用 **DISTINCT** 关键字：

```
sqlite> SELECT DISTINCT name FROM COMPANY;
```

这将产生以下结果，没有任何重复的条目：

```
NAME
----------
Paul
Allen
Teddy
Mark
David
Kim
James
```

SQLite 基础

SQLite PRAGMA

SQLite 的 **PRAGMA** 命令是一个特殊的命令，可以用在 SQLite 环境内控制各种环境变量和状态标志。一个 PRAGMA 值可以被读取，也可以根据需求进行设置。

语法

要查询当前的 PRAGMA 值，只需要提供该 pragma 的名字：

```
PRAGMA pragma_name;
```

要为 PRAGMA 设置一个新的值，语法如下：

```
PRAGMA pragma_name = value;
```

设置模式，可以是名称或等值的整数，但返回的值将始终是一个整数。

auto_vacuum Pragma

auto_vacuum Pragma 获取或设置 auto-vacuum 模式。语法如下：

```
PRAGMA [database.]auto_vacuum;
PRAGMA [database.]auto_vacuum = mode;
```

其中，**mode** 可以是以下任何一种：

Pragma 值	描述
0 或 NONE	禁用 Auto-vacuum。这是默认模式，意味着数据库文件尺寸大小不会缩小，除非手动使用 VACUUM 命令。
1 或 FULL	启用 Auto-vacuum，是全自动的。在该模式下，允许数据库文件随着数据从数据库移除而缩小。
2 或 INCREMENTAL	启用 Auto-vacuum，但是必须手动激活。在该模式下，引用数据被维持，免费页面只放在免费列表中。这些页面可在任何时候使用 **incremental_vacuum pragma** 进行覆盖。

cache_size Pragma

cache_size Pragma 可获取或暂时设置在内存中页面缓存的最大尺寸。语法如下：

```
PRAGMA [database.]cache_size;
PRAGMA [database.]cache_size = pages;
```

pages 值表示在缓存中的页面数。内置页面缓存的默认大小为 2,000 页，最小尺寸为 10 页。

case_sensitive_like Pragma

case_sensitive_like Pragma 控制内置的 LIKE 表达式的大小写敏感度。默认情况下，该 Pragma 为 false，这意味着，内置的 LIKE 操作符忽略字母的大小写。语法如下：

```
PRAGMA case_sensitive_like = [true|false];
```

目前没有办法查询该 Pragma 的当前状态。

count_changes Pragma

count_changes Pragma 获取或设置数据操作语句的返回值，如 INSERT、UPDATE 和 DELETE。语法如下：

```
PRAGMA count_changes;
PRAGMA count_changes = [true|false];
```

默认情况下，该 Pragma 为 false，这些语句不返回任何东西。如果设置为 true，每个所提到的语句将返回一个单行单列的表，由一个单一的整数值组成，该整数表示操作影响的行。

database_list Pragma

database_list Pragma 将用于列出了所有的数据库连接。语法如下：

```
PRAGMA database_list;
```

该 Pragma 将返回一个单行三列的表格，每当打开或附加数据库时，会给出数据库中的序列号，它的名称和相关的文件。

encoding Pragma

encoding Pragma 控制字符串如何编码及存储在数据库文件中。语法如下：

```
PRAGMA encoding;
PRAGMA encoding = format;
```

格式值可以是 UTF-8、UTF-16le 或 UTF-16be 之一。

freelist_count Pragma

freelist_count Pragma 返回一个整数，表示当前被标记为免费和可用的数据库页数。语法如下：

```
PRAGMA [database.]freelist_count;
```

格式值可以是 UTF-8、UTF-16le 或 UTF-16be 之一。

index_info Pragma

index_info Pragma 返回关于数据库索引的信息。语法如下：

```
PRAGMA [database.]index_info( index_name );
```

结果集将为每个包含在给出列序列的索引、表格内的列索引、列名称的列显示一行。

index_list Pragma

index_list Pragma 列出所有与表相关联的索引。语法如下：

```
PRAGMA [database.]index_list( table_name );
```

结果集将为每个给出列序列的索引、索引名称、表示索引是否唯一的标识显示一行。

journal_mode Pragma

journal_mode Pragma 获取或设置控制日志文件如何存储和处理的日志模式。语法如下：:

```
PRAGMA journal_mode;
PRAGMA journal_mode = mode;
PRAGMA database.journal_mode;
PRAGMA database.journal_mode = mode;
```

这里支持五种日志模式：

Pragma 值	描述
DELETE	默认模式。在该模式下，在事务结束时，日志文件将被删除。
TRUNCATE	日志文件被阶段为零字节长度。
PERSIST	日志文件被留在原地，但头部被重写，表明日志不再有效。
MEMORY	日志记录保留在内存中，而不是磁盘上。
OFF	不保留任何日志记录。

max_page_count Pragma

max_page_count Pragma 为数据库获取或设置允许的最大页数。语法如下：

```
PRAGMA [database.]max_page_count;
PRAGMA [database.]max_page_count = max_page;
```

默认值是 1,073,741,823，这是一个千兆的页面，即如果默认 1 KB 的页面大小，那么数据库中增长起来的一个兆字节。

page_count Pragma

page_count Pragma 返回当前数据库中的网页数量。语法如下：

```
PRAGMA [database.]page_count;
```

数据库文件的大小应该是 page_count * page_size。

page_size Pragma

page_size Pragma 获取或设置数据库页面的大小。语法如下：

```
PRAGMA [database.]page_size;
PRAGMA [database.]page_size = bytes;
```

默认情况下，允许的尺寸是 512、1024、2048、4096、8192、16384、32768 字节。改变现有数据库页面大小的唯一方法就是设置页面大小，然后立即 VACUUM 该数据库。

parser_trace Pragma

parser_trace Pragma 随着它解析 SQL 命令来控制打印的调试状态，语法如下：

```
PRAGMA parser_trace = [true|false];
```

默认情况下，它被设置为 false，但设置为 true 时则启用，此时 SQL 解析器会随着它解析 SQL 命令来打印出它的状态。

recursive_triggers Pragma

recursive_triggers Pragma 获取或设置递归触发器功能。如果未启用递归触发器，一个触发动作将不会触发另一个触发。语法如下：

```
PRAGMA recursive_triggers;
PRAGMA recursive_triggers = [true|false];
```

schema_version Pragma

schema_version Pragma 获取或设置存储在数据库头中的的架构版本值。语法如下：

```
PRAGMA [database.]schema_version;
PRAGMA [database.]schema_version = number;
```

这是一个 32 位有符号整数值，用来跟踪架构的变化。每当一个架构改变命令执行（比如 CREATE... 或 DROP...）时，这个值会递增。

secure_delete Pragma

secure_delete Pragma 用来控制内容是如何从数据库中删除。语法如下：

```
PRAGMA secure_delete;
PRAGMA secure_delete = [true|false];
PRAGMA database.secure_delete;
PRAGMA database.secure_delete = [true|false];
```

安全删除标志的默认值通常是关闭的，但是这是可以通过 SQLITE_SECURE_DELETE 构建选项来改变的。

sql_trace Pragma

sql_trace Pragma 用于把 SQL 跟踪结果转储到屏幕上。语法如下：

```
PRAGMA sql_trace;
PRAGMA sql_trace = [true|false];
```

SQLite 必须通过 SQLITE_DEBUG 指令来编译要引用的该 Pragma。

synchronous Pragma

synchronous Pragma 获取或设置当前磁盘的同步模式，该模式控制积极的 SQLite 如何将数据写入物理存储。语法如下：

```
PRAGMA [database.]synchronous;
PRAGMA [database.]synchronous = mode;
```

SQLite 支持下列同步模式：

Pragma 值	描述
0 或 OFF	不进行同步。
1 或 NORMAL	在关键的磁盘操作的每个序列后同步。
2 或 FULL	在每个关键的磁盘操作后同步。

temp_store Pragma

temp_store Pragma 获取或设置临时数据库文件所使用的存储模式。语法如下：

```
PRAGMA temp_store;
PRAGMA temp_store = mode;
```

SQLite 支持下列存储模式：

Pragma 值	描述
0 或 DEFAULT	默认使用编译时的模式。通常是 FILE。
1 或 FILE	使用基于文件的存储。
2 或 MEMORY	使用基于内存的存储。

temp_store_directory Pragma

temp_store_directory Pragma 获取或设置用于临时数据库文件的位置。语法如下：

```
PRAGMA temp_store_directory;
PRAGMA temp_store_directory = 'directory_path';
```

user_version Pragma

user_version Pragma 获取或设置存储在数据库头的用户自定义的版本值。语法如下：

```
PRAGMA [database.]user_version;
PRAGMA [database.]user_version = number;
```

这是一个 32 位的有符号整数值，可以由开发人员设置，用于版本跟踪的目的。

writable_schema Pragma

writable_schema Pragma 获取或设置是否能够修改系统表。语法如下：

```
PRAGMA writable_schema;
PRAGMA writable_schema = [true|false];
```

如果设置了该 Pragma，则表以 sqlite_ 开始，可以创建和修改，包括 sqlite_master 表。使用该 Pragma 时要注意，因为它可能导致整个数据库损坏。

SQLite 约束

约束是在表的数据列上强制执行的规则。这些是用来限制可以插入到表中的数据类型。这确保了数据库中数据的准确性和可靠性。

约束可以是列级或表级。列级约束仅适用于列，表级约束被应用到整个表。

以下是在 SQLite 中常用的约束。

- **NOT NULL** 约束：确保某列不能有 NULL 值。
- **DEFAULT** 约束：当某列没有指定值时，为该列提供默认值。
- **UNIQUE** 约束：确保某列中的所有值是不同的。
- **PRIMARY Key** 约束：唯一标识数据库表中的各行/记录。
- **CHECK** 约束：CHECK 约束确保某列中的所有值满足一定条件。

NOT NULL 约束

默认情况下，列可以保存 NULL 值。如果您不想某列有 NULL 值，那么需要在该列上定义此约束，指定在该列上不允许 NULL 值。

NULL 与没有数据是不一样的，它代表着未知的数据。

实例

例如，下面的 SQLite 语句创建一个新的表 COMPANY，并增加了五列，其中 ID、NAME 和 AGE 三列指定不接受 NULL 值：

```
CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
);
```

DEFAULT 约束

DEFAULT 约束在 INSERT INTO 语句没有提供一个特定的值时，为列提供一个默认值。

实例

例如，下面的 SQLite 语句创建一个新的表 COMPANY，并增加了五列。在这里，SALARY 列默认设置为 5000.00。所以当 INSERT INTO 语句没有为该列提供值时，该列将被设置为 5000.00。

```
CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL    DEFAULT 50000.00
);
```

UNIQUE 约束

UNIQUE 约束防止在一个特定的列存在两个记录具有相同的值。在 COMPANY 表中，例如，您可能要防止两个或两个以上的人具有相同的年龄。

实例

例如，下面的 SQLite 语句创建一个新的表 COMPANY，并增加了五列。在这里，AGE 列设置为 UNIQUE，所以不能有两个相同年龄的记录：

```
CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL UNIQUE,
   ADDRESS        CHAR(50),
   SALARY         REAL    DEFAULT 50000.00
);
```

PRIMARY KEY 约束

PRIMARY KEY 约束唯一标识数据库表中的每个记录。在一个表中可以有多个 UNIQUE 列，但只能有一个主键。在设计数据库表时，主键是很重要的。主键是唯一的 ID。

我们使用主键来引用表中的行。可通过把主键设置为其他表的外键，来创建表之间的关系。由于"长期存在编码监督"，在 SQLite 中，主键可以是 NULL，这是与其他数据库不同的地方。

主键是表中的一个字段，唯一标识数据库表中的各行/记录。主键必须包含唯一值。主键列不能有 NULL 值。

一个表只能有一个主键，它可以由一个或多个字段组成。当多个字段作为主键，它们被称为复合键。

如果一个表在任何字段上定义了一个主键，那么在这些字段上不能有两个记录具有相同的值。

实例

已经看到了我们创建以 ID 作为主键的 COMAPNY 表的各种实例：

```
CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
);
```

CHECK 约束

CHECK 约束启用输入一条记录要检查值的条件。如果条件值为 false，则记录违反了约束，且不能输入到表。

实例

例如，下面的 SQLite 创建一个新的表 COMPANY，并增加了五列。在这里，我们为 SALARY 列添加 CHECK，所以工资不能为零：

```
CREATE TABLE COMPANY3(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL    CHECK(SALARY > 0)
);
```

删除约束

SQLite 支持 ALTER TABLE 的有限子集。在 SQLite 中，ALTER TABLE 命令允许用户重命名表，或向现有表添加一个新的列。重命名列，删除一列，或从一个表中添加或删除约束都是不可能的。

SQLite Joins

SQLite 的 **Joins** 子句用于结合两个或多个数据库中表的记录。JOIN 是一种通过共同值来结合两个表中字段的手段。

SQL 定义了三种主要类型的连接：

- 交叉连接 - CROSS JOIN
- 内连接 - INNER JOIN
- 外连接 - OUTER JOIN

在我们继续之前，让我们假设有两个表 COMPANY 和 DEPARTMENT。我们已经看到了用来填充 COMPANY 表的 INSERT 语句。现在让我们假设 COMPANY 表的记录列表如下：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

另一个表是 DEPARTMENT，定义如下：

```
CREATE TABLE DEPARTMENT(
   ID INT PRIMARY KEY      NOT NULL,
   DEPT           CHAR(50) NOT NULL,
   EMP_ID         INT      NOT NULL
);
```

下面是填充 DEPARTMENT 表的 INSERT 语句：

```
INSERT INTO DEPARTMENT (ID, DEPT, EMP_ID)
VALUES (1, 'IT Billing', 1 );

INSERT INTO DEPARTMENT (ID, DEPT, EMP_ID)
VALUES (2, 'Engineering', 2 );

INSERT INTO DEPARTMENT (ID, DEPT, EMP_ID)
VALUES (3, 'Finance', 7 );
```

最后，我们在 DEPARTMENT 表中有下列的记录列表：

```
ID          DEPT        EMP_ID
----------  ----------  ----------
1           IT Billing  1
2           Engineerin  2
3           Finance     7
```

交叉连接 - CROSS JOIN

交叉连接（CROSS JOIN）把第一个表的每一行与第二个表的每一行进行匹配。如果两个输入表分别有 x 和 y 列，则结果表有 x+y 列。由于交叉连接（CROSS JOIN）有可能产生非常大的表，使用时必须谨慎，只在适当的时候使用它们。

下面是交叉连接（CROSS JOIN）的语法：

```
SELECT ... FROM table1 CROSS JOIN table2 ...
```

基于上面的表，我们可以写一个交叉连接（CROSS JOIN），如下所示：

```
sqlite> SELECT EMP_ID, NAME, DEPT FROM COMPANY CROSS JOIN DEPARTMEN
```

上面的查询会产生以下结果：

```
EMP_ID      NAME        DEPT
----------  ----------  ----------
1           Paul        IT Billing
2           Paul        Engineerin
7           Paul        Finance
1           Allen       IT Billing
2           Allen       Engineerin
7           Allen       Finance
1           Teddy       IT Billing
2           Teddy       Engineerin
7           Teddy       Finance
1           Mark        IT Billing
2           Mark        Engineerin
7           Mark        Finance
1           David       IT Billing
2           David       Engineerin
7           David       Finance
1           Kim         IT Billing
2           Kim         Engineerin
7           Kim         Finance
1           James       IT Billing
2           James       Engineerin
7           James       Finance
```

内连接 - INNER JOIN

内连接（INNER JOIN）根据连接谓词结合两个表（table1 和 table2）的列值来创建一个新的结果表。查询会把 table1 中的每一行与 table2 中的每一行进行比较，找到所有满足连接谓词的行的匹配对。当满足连接谓词时，A 和 B 行的每个匹配对的列值会合并成一个结果行。

内连接（INNER JOIN）是最常见的连接类型，是默认的连接类型。INNER 关键字是可选的。

下面是内连接（INNER JOIN）的语法：

```
SELECT ... FROM table1 [INNER] JOIN table2 ON conditional_expressio
```

为了避免冗余，并保持较短的措辞，可以使用 **USING** 表达式声明内连接（INNER JOIN）条件。这个表达式指定一个或多个列的列表：

```
SELECT ... FROM table1 JOIN table2 USING ( column1 ,... ) ...
```

自然连接（NATURAL JOIN）类似于 **JOIN...USING**，只是它会自动测试存在两个表中的每一列的值之间相等值：

```
SELECT ... FROM table1 NATURAL JOIN table2...
```

基于上面的表，我们可以写一个内连接（INNER JOIN），如下所示：

```
sqlite> SELECT EMP_ID, NAME, DEPT FROM COMPANY INNER JOIN DEPARTMEN
        ON COMPANY.ID = DEPARTMENT.EMP_ID;
```

上面的查询会产生以下结果：

```
EMP_ID      NAME        DEPT
----------  ----------  ----------
1           Paul        IT Billing
2           Allen       Engineerin
7           James       Finance
```

外连接 - OUTER JOIN

外连接（OUTER JOIN）是内连接（INNER JOIN）的扩展。虽然 SQL 标准定义了三种类型的外连接：LEFT、RIGHT、FULL，但 SQLite 只支持 左外连接（**LEFT OUTER JOIN**）。

外连接（OUTER JOIN）声明条件的方法与内连接（INNER JOIN）是相同的，使用 ON、USING 或 NATURAL 关键字来表达。最初的结果表以相同的方式进行计算。一旦主连接计算完成，外连接（OUTER JOIN）将从一个或两个表中任何未连接的行合并进来，外连接的列使用 NULL 值，将它们附加到结果表中。

下面是左外连接（LEFT OUTER JOIN）的语法：

```
SELECT ... FROM table1 LEFT OUTER JOIN table2 ON conditional_expres
```

为了避免冗余，并保持较短的措辞，可以使用 **USING** 表达式声明外连接（OUTER JOIN）条件。这个表达式指定一个或多个列的列表：

```
SELECT ... FROM table1 LEFT OUTER JOIN table2 USING ( column1 ,...
```

基于上面的表，我们可以写一个外连接（OUTER JOIN），如下所示：

```
sqlite> SELECT EMP_ID, NAME, DEPT FROM COMPANY LEFT OUTER JOIN DEPA
        ON COMPANY.ID = DEPARTMENT.EMP_ID;
```

上面的查询会产生以下结果：

```
EMP_ID      NAME        DEPT
----------  ----------  ----------
1           Paul        IT Billing
2           Allen       Engineerin
            Teddy
            Mark
            David
            Kim
7           James       Finance
```

SQLite Unions 子句

SQLite的 **UNION** 子句/运算符用于合并两个或多个 SELECT 语句的结果，不返回任何重复的行。

为了使用 UNION，每个 SELECT 被选择的列数必须是相同的，相同数目的列表达式，相同的数据类型，并确保它们有相同的顺序，但它们不必具有相同的长度。

语法

UNION 的基本语法如下：

```
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]

UNION

SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]
```

这里给定的条件根据需要可以是任何表达式。

实例

假设有下面两个表，（1）COMPANY 表如下所示：

```
sqlite> select * from COMPANY;
ID          NAME                  AGE         ADDRESS     SALARY
----------  --------------------  ----------  ----------  ---------
1           Paul                  32          California  20000.0
2           Allen                 25          Texas       15000.0
3           Teddy                 23          Norway      20000.0
4           Mark                  25          Rich-Mond   65000.0
5           David                 27          Texas       85000.0
6           Kim                   22          South-Hall  45000.0
7           James                 24          Houston     10000.0
```

（2）另一个表是 DEPARTMENT，如下所示：

```
ID          DEPT                  EMP_ID
----------  --------------------  ----------
1           IT Billing            1
2           Engineering           2
3           Finance               7
4           Engineering           3
5           Finance               4
6           Engineering           5
7           Finance               6
```

现在，让我们使用 SELECT 语句及 UNION 子句来连接两个表，如下所示：

```
sqlite> SELECT EMP_ID, NAME, DEPT FROM COMPANY INNER JOIN DEPARTMEN
        ON COMPANY.ID = DEPARTMENT.EMP_ID
   UNION
     SELECT EMP_ID, NAME, DEPT FROM COMPANY LEFT OUTER JOIN DEPARTM
        ON COMPANY.ID = DEPARTMENT.EMP_ID;
```

这将产生以下结果：

```
EMP_ID      NAME                  DEPT
----------  --------------------  ----------
1           Paul                  IT Billing
2           Allen                 Engineerin
3           Teddy                 Engineerin
4           Mark                  Finance
5           David                 Engineerin
6           Kim                   Finance
7           James                 Finance
```

UNION ALL 子句

UNION ALL 运算符用于结合两个 SELECT 语句的结果，包括重复行。

适用于 UNION 的规则同样适用于 UNION ALL 运算符。

语法

UNION ALL 的基本语法如下：

```
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]

UNION ALL

SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]
```

这里给定的条件根据需要可以是任何表达式。

实例

现在，让我们使用 SELECT 语句及 UNION ALL 子句来连接两个表，如下所示：

```
sqlite> SELECT EMP_ID, NAME, DEPT FROM COMPANY INNER JOIN DEPARTMEN
        ON COMPANY.ID = DEPARTMENT.EMP_ID
   UNION ALL
     SELECT EMP_ID, NAME, DEPT FROM COMPANY LEFT OUTER JOIN DEPARTM
        ON COMPANY.ID = DEPARTMENT.EMP_ID;
```

这将产生以下结果：

```
EMP_ID      NAME                  DEPT
----------  --------------------  ----------
1           Paul                  IT Billing
2           Allen                 Engineerin
3           Teddy                 Engineerin
4           Mark                  Finance
5           David                 Engineerin
6           Kim                   Finance
7           James                 Finance
1           Paul                  IT Billing
2           Allen                 Engineerin
3           Teddy                 Engineerin
4           Mark                  Finance
5           David                 Engineerin
6           Kim                   Finance
7           James                 Finance
```

SQLite NULL 值

SQLite 的 **NULL** 是用来表示一个缺失值的项。表中的一个 NULL 值是在字段中显示为空白的一个值。

带有 NULL 值的字段是一个不带有值的字段。NULL 值与零值或包含空格的字段是不同的，理解这点是非常重要的。

语法

创建表时使用 **NULL** 的基本语法如下：

```
SQLite> CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
);
```

在这里，**NOT NULL** 表示列总是接受给定数据类型的显式值。这里有两个列我们没有使用 NOT NULL，这意味着这两个列不能为 NULL。

带有 NULL 值的字段在记录创建的时候可以保留为空。

实例

NULL 值在选择数据时会引起问题，因为当把一个未知的值与另一个值进行比较时，结果总是未知的，且不会包含在最后的结果中。假设有下面的表，COMPANY 的记录如下所示：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

让我们使用 UPDATE 语句来设置一些允许空值的值为 NULL，如下所示：

```
sqlite> UPDATE COMPANY SET ADDRESS = NULL, SALARY = NULL where ID I
```

现在，COMPANY 表的记录如下所示：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22
7           James       24
```

接下来，让我们看看 **IS NOT NULL** 运算符的用法，它用来列出所有 SALARY 不为 NULL 的记录：

```
sqlite> SELECT  ID, NAME, AGE, ADDRESS, SALARY
        FROM COMPANY
        WHERE SALARY IS NOT NULL;
```

上面的 SQLite 语句将产生下面的结果：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

下面是 **IS NULL** 运算符的用法，将列出所有 SALARY 为 NULL 的记录：

```
sqlite> SELECT  ID, NAME, AGE, ADDRESS, SALARY
        FROM COMPANY
        WHERE SALARY IS NULL;
```

上面的 SQLite 语句将产生下面的结果：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
6           Kim         22
7           James       24
```

SQLite 别名

您可以暂时把表或列重命名为另一个名字，这被称为别名。使用表别名是指在一个特定的 SQLite 语句中重命名表。重命名是临时的改变，在数据库中实际的表的名称不会改变。

列别名用来为某个特定的 SQLite 语句重命名表中的列。

语法

表 别名的基本语法如下：

```
SELECT column1, column2....
FROM table_name AS alias_name
WHERE [condition];
```

列 别名的基本语法如下：

```
SELECT column_name AS alias_name
FROM table_name
WHERE [condition];
```

实例

假设有下面两个表，（1）COMPANY 表如下所示：

```
sqlite> select * from COMPANY;
ID          NAME                  AGE         ADDRESS     SALARY
----------  --------------------  ----------  ----------  ----------
1           Paul                  32          California  20000.0
2           Allen                 25          Texas       15000.0
3           Teddy                 23          Norway      20000.0
4           Mark                  25          Rich-Mond   65000.0
5           David                 27          Texas       85000.0
6           Kim                   22          South-Hall  45000.0
7           James                 24          Houston     10000.0
```

（2）另一个表是 DEPARTMENT，如下所示：

```
ID          DEPT                  EMP_ID
----------  --------------------  ----------
1           IT Billing            1
2           Engineering           2
3           Finance               7
4           Engineering           3
5           Finance               4
6           Engineering           5
7           Finance               6
```

现在，下面是 表别名 的用法，在这里我们使用 C 和 D 分别作为 COMPANY 和 DEPARTMENT 表的别名：

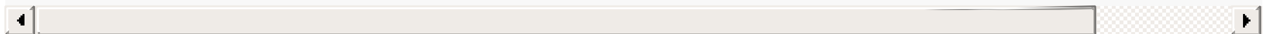
```
sqlite> SELECT C.ID, C.NAME, C.AGE, D.DEPT
        FROM COMPANY AS C, DEPARTMENT AS D
        WHERE  C.ID = D.EMP_ID;
```

上面的 SQLite 语句将产生下面的结果：

```
ID          NAME        AGE         DEPT
----------  ----------  ----------  ----------
1           Paul        32          IT Billing
2           Allen       25          Engineerin
3           Teddy       23          Engineerin
4           Mark        25          Finance
5           David       27          Engineerin
6           Kim         22          Finance
7           James       24          Finance
```

让我们看一个 列别名 的实例，在这里 COMPANY_ID 是 ID 列的别名，COMPANY_NAME 是 name 列的别名：

```
sqlite> SELECT C.ID AS COMPANY_ID, C.NAME AS COMPANY_NAME, C.AGE, D
        FROM COMPANY AS C, DEPARTMENT AS D
        WHERE  C.ID = D.EMP_ID;
```



上面的 SQLite 语句将产生下面的结果：

```
COMPANY_ID  COMPANY_NAME  AGE         DEPT
----------  ------------  ----------  ----------
1           Paul          32          IT Billing
2           Allen         25          Engineerin
3           Teddy         23          Engineerin
4           Mark          25          Finance
5           David         27          Engineerin
6           Kim           22          Finance
7           James         24          Finance
```

SQLite 触发器（Trigger）

SQLite 的触发器是数据库的回调函数，它会自动执行/指定的数据库事件发生时调用。以下是关于SQLite的触发器的要点： SQLite 触发器（**Trigger**）是数据库的回调函数，它会在指定的数据库事件发生时自动执行/调用。以下是关于 SQLite 的触发器（Trigger）的要点：

- SQLite 的触发器（Trigger）可以指定在特定的数据库表发生 DELETE、INSERT 或 UPDATE 时触发，或在一个或多个指定表的列发生更新时触发。
- SQLite 只支持 FOR EACH ROW 触发器（Trigger），没有 FOR EACH STATEMENT 触发器（Trigger）。因此，明确指定 FOR EACH ROW 是可选的。
- WHEN 子句和触发器（Trigger）动作可能访问使用表单 **NEW.column-name** 和 **OLD.column-name** 的引用插入、删除或更新的行元素，其中 column-name 是从与触发器关联的表的列的名称。
- 如果提供 WHEN 子句，则只针对 WHEN 子句为真的指定行执行 SQL 语句。如果没有提供 WHEN 子句，则针对所有行执行 SQL 语句。
- BEFORE 或 AFTER 关键字决定何时执行触发器动作，决定是在关联行的插入、修改或删除之前或者之后执行触发器动作。
- 当触发器相关联的表删除时，自动删除触发器（Trigger）。
- 要修改的表必须存在于同一数据库中，作为触发器被附加的表或视图，且必须只使用 **tablename**，而不是 **database.tablename**。
- 一个特殊的 SQL 函数 RAISE() 可用于触发器程序内抛出异常。

语法

创建 触发器（**Trigger**） 的基本语法如下：

```
CREATE  TRIGGER trigger_name [BEFORE|AFTER] event_name
ON table_name
BEGIN
 -- Trigger logic goes here....
END;
```

在这里，**event_name** 可以是在所提到的表 **table_name** 上的 *INSERT*、*DELETE* 和 *UPDATE* 数据库操作。您可以在表名后选择指定 FOR EACH ROW。

以下是在 UPDATE 操作上在表的一个或多个指定列上创建触发器（Trigger）的语法：

```
CREATE  TRIGGER trigger_name [BEFORE|AFTER] UPDATE OF column_name
ON table_name
BEGIN
 -- Trigger logic goes here....
END;
```

实例

让我们假设一个情况，我们要为被插入到新创建的 COMPANY 表（如果已经存在，则删除重新创建）中的每一个记录保持审计试验：

```
sqlite> CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
);
```

为了保持审计试验，我们将创建一个名为 AUDIT 的新表。每当 COMPANY 表中有一个新的记录项时，日志消息将被插入其中：

```
sqlite> CREATE TABLE AUDIT(
    EMP_ID INT NOT NULL,
    ENTRY_DATE TEXT NOT NULL
);
```

在这里，ID 是 AUDIT 记录的 ID，EMP_ID 是来自 COMPANY 表的 ID，DATE 将保持 COMPANY 中记录被创建时的时间戳。所以，现在让我们在 COMPANY 表上创建一个触发器，如下所示：

```
sqlite> CREATE TRIGGER audit_log AFTER INSERT
ON COMPANY
BEGIN
   INSERT INTO AUDIT(EMP_ID, ENTRY_DATE) VALUES (new.ID, datetime(
END;
```

现在，我们将开始在 COMPANY 表中插入记录，这将导致在 AUDIT 表中创建一个审计日志记录。因此，让我们在 COMPANY 表中创建一个记录，如下所示：

```
sqlite> INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Paul', 32, 'California', 20000.00 );
```

这将在 COMPANY 表中创建如下一个记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
```

同时，将在 AUDIT 表中创建一个记录。这个纪录是触发器的结果，这是我们在 COMPANY 表上的 INSERT 操作上创建的触发器（Trigger）。类似的，可以根据需要在 UPDATE 和 DELETE 操作上创建触发器（Trigger）。

```
EMP_ID      ENTRY_DATE
----------  -------------------
1           2013-04-05 06:26:00
```

列出触发器（TRIGGERS）

您可以从 **sqlite_master** 表中列出所有触发器，如下所示：

```
sqlite> SELECT name FROM sqlite_master
WHERE type = 'trigger';
```

上面的 SQLite 语句只会列出一个条目，如下：

```
name
----------
audit_log
```

如果您想要列出特定表上的触发器，则使用 AND 子句连接表名，如下所示：

```
sqlite> SELECT name FROM sqlite_master
WHERE type = 'trigger' AND tbl_name = 'COMPANY';
```

上面的 SQLite 语句只会列出一个条目，如下：

```
name
----------
audit_log
```

删除触发器（TRIGGERS）

下面是 DROP 命令，可用于删除已有的触发器：

```
sqlite> DROP TRIGGER trigger_name;
```

SQLite 索引（Index）

索引（Index）是一种特殊的查找表，数据库搜索引擎用来加快数据检索。简单地说，索引是一个指向表中数据的指针。一个数据库中的索引与一本书后边的索引是非常相似的。

例如，如果您想在一本讨论某个话题的书中引用所有页面，您首先需要指向索引，索引按字母顺序列出了所有主题，然后指向一个或多个特定的页码。

索引有助于加快 SELECT 查询和 WHERE 子句，但它会减慢使用 UPDATE 和 INSERT 语句时的数据输入。索引可以创建或删除，但不会影响数据。

使用 CREATE INDEX 语句创建索引，它允许命名索引，指定表及要索引的一列或多列，并指示索引是升序排列还是降序排列。

索引也可以是唯一的，与 UNIQUE 约束类似，在列上或列组合上防止重复条目。

CREATE INDEX 命令

CREATE INDEX 的基本语法如下：

```
CREATE INDEX index_name ON table_name;
```

单列索引

单列索引是一个只基于表的一个列上创建的索引。基本语法如下：

```
CREATE INDEX index_name
ON table_name (column_name);
```

唯一索引

使用唯一索引不仅是为了性能，同时也为了数据的完整性。唯一索引不允许任何重复的值插入到表中。基本语法如下：

```
CREATE INDEX index_name
on table_name (column_name);
```

组合索引

组合索引是基于一个表的两个或多个列上创建的索引。基本语法如下：

```
CREATE INDEX index_name
on table_name (column1, column2);
```

是否要创建一个单列索引还是组合索引，要考虑到您在作为查询过滤条件的 WHERE 子句中使用非常频繁的列。

如果值使用到一个列，则选择使用单列索引。如果在作为过滤的 WHERE 子句中有两个或多个列经常使用，则选择使用组合索引。

隐式索引

隐式索引是在创建对象时，由数据库服务器自动创建的索引。索引自动创建为主键约束和唯一约束。

实例

下面是一个例子，我们将在 COMPANY 表的 salary 列上创建一个索引：

```
sqlite> CREATE INDEX salary_index ON COMPANY (salary);
```

现在，让我们使用 **.indices** 命令列出 COMPANY 表上所有可用的索引，如下所示：

```
sqlite> .indices COMPANY
```

这将产生如下结果，其中 *sqlite_autoindex_COMPANY_1* 是创建表时创建的隐式索引。

```
salary_index
sqlite_autoindex_COMPANY_1
```

您可以列出数据库范围的所有索引，如下所示：

```
sqlite> SELECT * FROM sqlite_master WHERE type = 'index';
```

DROP INDEX 命令

一个索引可以使用 SQLite 的 **DROP** 命令删除。当删除索引时应特别注意，因为性能可能会下降或提高。

基本语法如下：

```
DROP INDEX index_name;
```

您可以使用下面的语句来删除之前创建的索引：

```
sqlite> DROP INDEX salary_index;
```

什么情况下要避免使用索引？

虽然索引的目的在于提高数据库的性能，但这里有几个情况需要避免使用索引。使用索引时，应重新考虑下列准则：

- 索引不应该使用在较小的表上。
- 索引不应该使用在有频繁的大批量的更新或插入操作的表上。
- 索引不应该使用在含有大量的 NULL 值的列上。
- 索引不应该使用在频繁操作的列上。

SQLite Indexed By

"INDEXED BY index-name" 子句规定必须需要命名的索引来查找前面表中值。

如果索引名 index-name 不存在或不能用于查询，然后 SQLite 语句的准备失败。

"NOT INDEXED" 子句规定当访问前面的表（包括由 UNIQUE 和 PRIMARY KEY 约束创建的隐式索引）时，没有使用索引。

然而，即使指定了 "NOT INDEXED"，INTEGER PRIMARY KEY 仍然可以被用于查找条目。

语法

下面是 INDEXED BY 子句的语法，它可以与 DELETE、UPDATE 或 SELECT 语句一起使用：

```
SELECT|DELETE|UPDATE column1, column2...
INDEXED BY (index_name)
table_name
WHERE (CONDITION);
```

实例

假设有表 COMPANY，我们将创建一个索引，并用它进行 INDEXED BY 操作。

```
sqlite> CREATE INDEX salary_index ON COMPANY(salary);
sqlite>
```

现在使用 INDEXED BY 子句从表 COMPANY 中选择数据，如下所示：

```
sqlite> SELECT * FROM COMPANY INDEXED BY salary_index WHERE salary
```

SQLite Alter 命令

SQLite 的 **ALTER TABLE** 命令不通过执行一个完整的转储和数据的重载来修改已有的表。您可以使用 ALTER TABLE 语句重命名表，使用 ALTER TABLE 语句还可以在已有的表中添加额外的列。

在 SQLite 中，除了重命名表和在已有的表中添加列，ALTER TABLE 命令不支持其他操作。

语法

用来重命名已有的表的 **ALTER TABLE** 的基本语法如下：

```
ALTER TABLE database_name.table_name RENAME TO new_table_name;
```

用来在已有的表中添加一个新的列的 **ALTER TABLE** 的基本语法如下：

```
ALTER TABLE database_name.table_name ADD COLUMN column_def...;
```

实例

假设我们的 COMPANY 表有如下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

现在，让我们尝试使用 ALTER TABLE 语句重命名该表，如下所示：

```
sqlite> ALTER TABLE COMPANY RENAME TO OLD_COMPANY;
```

上面的 SQLite 语句将重命名 COMPANY 表为 OLD_COMPANY。现在，让我们尝试在 OLD_COMPANY 表中添加一个新的列，如下所示：

```
sqlite> ALTER TABLE OLD_COMPANY ADD COLUMN SEX char(1);
```

现在，COMPANY 表已经改变，使用 SELECT 语句输出如下：

```
ID          NAME        AGE         ADDRESS     SALARY      SEX
----------  ----------  ----------  ----------  ----------  ---
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

请注意，新添加的列是以 NULL 值来填充的。

SQLite Truncate Table

在 SQLite 中，并没有 TRUNCATE TABLE 命令，但可以使用 SQLite 的 **DELETE** 命令从已有的表中删除全部的数据，但建议使用 DROP TABLE 命令删除整个表，然后再重新创建一遍。

语法

DELETE 命令的基本语法如下：

```
sqlite> DELETE FROM table_name;
```

DROP TABLE 的基本语法如下：

```
sqlite> DROP TABLE table_name;
```

如果您使用 DELETE TABLE 命令删除所有记录，建议使用 **VACUUM** 命令清除未使用的空间。

实例

假设 COMPANY 表有如下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面为删除上表记录的实例：

```
SQLite> DELETE FROM COMPANY;
SQLite> VACUUM;
```

现在，COMPANY 表中的记录完全被删除，使用 SELECT 语句将没有任何输出。

SQLite 视图（View）

视图（View）只不过是通过相关的名称存储在数据库中的一个 SQLite 语句。视图（View）实际上是一个以预定义的 SQLite 查询形式存在的表的组合。

视图（View）可以包含一个表的所有行或从一个或多个表选定行。视图（View）可以从一个或多个表创建，这取决于要创建视图的 SQLite 查询。、

视图（View）是一种虚表，允许用户实现以下几点：

- 用户或用户组查找结构数据的方式更自然或直观。
- 限制数据访问，用户只能看到有限的数据，而不是完整的表。
- 汇总各种表中的数据，用于生成报告。

SQLite 视图是只读的，因此可能无法在视图上执行 DELETE、INSERT 或 UPDATE 语句。但是可以在视图上创建一个触发器，当尝试 DELETE、INSERT 或 UPDATE 视图时触发，需要做的动作在触发器内容中定义。

创建视图

SQLite 的视图是使用 **CREATE VIEW** 语句创建的。SQLite 视图可以从一个单一的表、多个表或其他视图创建。

CREATE VIEW 的基本语法如下：

```
CREATE [TEMP | TEMPORARY] VIEW view_name AS
SELECT column1, column2.....
FROM table_name
WHERE [condition];
```

您可以在 SELECT 语句中包含多个表，这与在正常的 SQL SELECT 查询中的方式非常相似。如果使用了可选的 TEMP 或 TEMPORARY 关键字，则将在临时数据库中创建视图。

实例

假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

现在，下面是一个从 COMPANY 表创建视图的实例。视图只从 COMPANY 表中选取几列：

```
sqlite> CREATE VIEW COMPANY_VIEW AS
SELECT ID, NAME, AGE
FROM  COMPANY;
```

现在，可以查询 COMPANY_VIEW，与查询实际表的方式类似。下面是实例：

```
sqlite> SELECT * FROM COMPANY_VIEW;
```

这将产生以下结果:

```
ID          NAME        AGE
----------  ----------  ----------
1           Paul        32
2           Allen       25
3           Teddy       23
4           Mark        25
5           David       27
6           Kim         22
7           James       24
```

删除视图

要删除视图，只需使用带有 **view_name** 的 DROP VIEW 语句。DROP VIEW 的基本语法如下：

```
sqlite> DROP VIEW view_name;
```

下面的命令将删除我们在前面创建的 COMPANY_VIEW 视图：

```
sqlite> DROP VIEW COMPANY_VIEW;
```

SQLite 事务（Transaction）

事务（Transaction）是一个对数据库执行工作单元。事务（Transaction）是以逻辑顺序完成的工作单位或序列，可以是由用户手动操作完成，也可以是由某种数据库程序自动完成。

事务（Transaction）是指一个或多个更改数据库的扩展。例如，如果您正在创建一个记录或者更新一个记录或者从表中删除一个记录，那么您正在该表上执行事务。重要的是要控制事务以确保数据的完整性和处理数据库错误。

实际上，您可以把许多的 SQLite 查询联合成一组，把所有这些放在一起作为事务的一部分进行执行。

事务的属性

事务（Transaction）具有以下四个标准属性，通常根据首字母缩写为 ACID：

- 原子性（**Atomicity**）：确保工作单位内的所有操作都成功完成，否则，事务会在出现故障时终止，之前的操作也会回滚到以前的状态。
- 一致性（**Consistency)**：确保数据库在成功提交的事务上正确地改变状态。
- 隔离性（**Isolation**）：使事务操作相互独立和透明。
- 持久性（**Durability**）：确保已提交事务的结果或效果在系统发生故障的情况下仍然存在。

事务控制

使用下面的命令来控制事务：

- **BEGIN TRANSACTION**：开始事务处理。
- **COMMIT**：保存更改，或者可以使用 **END TRANSACTION** 命令。
- **ROLLBACK**：回滚所做的更改。

事务控制命令只与 DML 命令 INSERT、UPDATE 和 DELETE 一起使用。他们不能在创建表或删除表时使用，因为这些操作在数据库中是自动提交的。

BEGIN TRANSACTION 命令

事务（Transaction）可以使用 BEGIN TRANSACTION 命令或简单的 BEGIN 命令来启动。此类事务通常会持续执行下去，直到遇到下一个 COMMIT 或 ROLLBACK 命令。不过在数据库关闭或发生错误时，事务处理也会回滚。以下是启动一个事务

的简单语法：

```
BEGIN;

or

BEGIN TRANSACTION;
```

COMMIT 命令

COMMIT 命令是用于把事务调用的更改保存到数据库中的事务命令。

COMMIT 命令把自上次 COMMIT 或 ROLLBACK 命令以来的所有事务保存到数据库。

COMMIT 命令的语法如下：

```
COMMIT;

or

END TRANSACTION;
```

ROLLBACK 命令

ROLLBACK 命令是用于撤消尚未保存到数据库的事务的事务命令。

ROLLBACK 命令只能用于撤销自上次发出 COMMIT 或 ROLLBACK 命令以来的事务。

ROLLBACK 命令的语法如下：

```
ROLLBACK;
```

实例

假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

现在，让我们开始一个事务，并从表中删除 age = 25 的记录，最后，我们使用 ROLLBACK 命令撤消所有的更改。

```
sqlite> BEGIN;
sqlite> DELETE FROM COMPANY WHERE AGE = 25;
sqlite> ROLLBACK;
```

检查 COMPANY 表，仍然有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

现在，让我们开始另一个事务，从表中删除 age = 25 的记录，最后我们使用 COMMIT 命令提交所有的更改。

```
sqlite> BEGIN;
sqlite> DELETE FROM COMPANY WHERE AGE = 25;
sqlite> COMMIT;
```

检查 COMPANY 表，有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
3           Teddy       23          Norway      20000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

SQLite 子查询

子查询或内部查询或嵌套查询是在另一个 SQLite 查询内嵌入在 WHERE 子句中的查询。

使用子查询返回的数据将被用在主查询中作为条件，以进一步限制要检索的数据。

子查询可以与 SELECT、INSERT、UPDATE 和 DELETE 语句一起使用，可伴随着使用运算符如 =、<、>、>=、<=、IN、BETWEEN 等。

以下是子查询必须遵循的几个规则：

- 子查询必须用括号括起来。
- 子查询在 SELECT 子句中只能有一个列，除非在主查询中有多列，与子查询的所选列进行比较。
- ORDER BY 不能用在子查询中，虽然主查询可以使用 ORDER BY。可以在子查询中使用 GROUP BY，功能与 ORDER BY 相同。
- 子查询返回多于一行，只能与多值运算符一起使用，如 IN 运算符。
- BETWEEN 运算符不能与子查询一起使用，但是，BETWEEN 可在子查询内使用。

SELECT 语句中的子查询使用

子查询通常与 SELECT 语句一起使用。基本语法如下：

```
SELECT column_name [, column_name ]
FROM   table1 [, table2 ]
WHERE  column_name OPERATOR
      (SELECT column_name [, column_name ]
      FROM table1 [, table2 ]
      [WHERE])
```

实例

假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

现在，让我们检查 SELECT 语句中的子查询使用：

```
sqlite> SELECT *
     FROM COMPANY
     WHERE ID IN (SELECT ID
                  FROM COMPANY
                  WHERE SALARY > 45000) ;
```

这将产生以下结果:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

INSERT 语句中的子查询使用

子查询也可以与 INSERT 语句一起使用。INSERT 语句使用子查询返回的数据插入到另一个表中。在子查询中所选择的数据可以用任何字符、日期或数字函数修改。

基本语法如下：

```
INSERT INTO table_name [ (column1 [, column2 ]) ]
           SELECT [ *|column1 [, column2 ]
           FROM table1 [, table2 ]
           [ WHERE VALUE OPERATOR ]
```

实例

假设 COMPANY_BKP 的结构与 COMPANY 表相似，且可使用相同的 CREATE TABLE 进行创建，只是表名改为 COMPANY_BKP。现在把整个 COMPANY 表复制到 COMPANY_BKP，语法如下：

```
sqlite> INSERT INTO COMPANY_BKP
     SELECT * FROM COMPANY
     WHERE ID IN (SELECT ID
                  FROM COMPANY) ;
```

UPDATE 语句中的子查询使用

子查询可以与 UPDATE 语句结合使用。当通过 UPDATE 语句使用子查询时，表中单个或多个列被更新。

基本语法如下：

```
UPDATE table
SET column_name = new_value
[ WHERE OPERATOR [ VALUE ]
   (SELECT COLUMN_NAME
   FROM TABLE_NAME)
   [ WHERE) ]
```

实例

假设，我们有 COMPANY_BKP 表，是 COMPANY 表的备份。

下面的实例把 COMPANY 表中所有 AGE 大于或等于 27 的客户的 SALARY 更新为原来的 0.50 倍：

```
sqlite> UPDATE COMPANY
     SET SALARY = SALARY * 0.50
     WHERE AGE IN (SELECT AGE FROM COMPANY_BKP
                   WHERE AGE >= 27 );
```

这将影响两行，最后 COMPANY 表中的记录如下：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  10000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       42500.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

DELETE 语句中的子查询使用

子查询可以与 DELETE 语句结合使用，就像上面提到的其他语句一样。

基本语法如下：

```
DELETE FROM TABLE_NAME
[ WHERE OPERATOR [ VALUE ]
   (SELECT COLUMN_NAME
   FROM TABLE_NAME)
   [ WHERE) ]
```

实例

假设，我们有 COMPANY_BKP 表，是 COMPANY 表的备份。

下面的实例删除 COMPANY 表中所有 AGE 大于或等于 27 的客户记录：

```
sqlite> DELETE FROM COMPANY
     WHERE AGE IN (SELECT AGE FROM COMPANY_BKP
                   WHERE AGE > 27 );
```

这将影响两行，最后 COMPANY 表中的记录如下：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       42500.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

SQLite Autoincrement（自动递增）

SQLite 的 **AUTOINCREMENT** 是一个关键字，用于表中的字段值自动递增。我们可以在创建表时在特定的列名称上使用 **AUTOINCREMENT** 关键字实现该字段值的自动增加。

关键字 **AUTOINCREMENT** 只能用于整型（INTEGER）字段。

语法

AUTOINCREMENT 关键字的基本用法如下：

```
CREATE TABLE table_name(
   column1 INTEGER AUTOINCREMENT,
   column2 datatype,
   column3 datatype,
   .....
   columnN datatype,
);
```

实例

假设要创建的 COMPANY 表如下所示：

```
sqlite> CREATE TABLE COMPANY(
   ID INTEGER PRIMARY KEY   AUTOINCREMENT,
   NAME           TEXT      NOT NULL,
   AGE            INT       NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
);
```

现在，向 COMPANY 表插入以下记录：

```
INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)
VALUES ( 'Paul', 32, 'California', 20000.00 );

INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)
VALUES ('Allen', 25, 'Texas', 15000.00 );

INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)
VALUES ('Teddy', 23, 'Norway', 20000.00 );

INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)
VALUES ( 'Mark', 25, 'Rich-Mond ', 65000.00 );

INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)
VALUES ( 'David', 27, 'Texas', 85000.00 );

INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)
VALUES ( 'Kim', 22, 'South-Hall', 45000.00 );

INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)
VALUES ( 'James', 24, 'Houston', 10000.00 );
```

这将向 COMPANY 表插入 7 个元组，此时 COMPANY 表的记录如下：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

SQLite 注入

如果您的站点允许用户通过网页输入，并将输入内容插入到 SQLite 数据库中，这个时候您就面临着一个被称为 SQL 注入的安全问题。本章节将向您讲解如何防止这种情况的发生，确保脚本和 SQLite 语句的安全。

注入通常在请求用户输入时发生，比如需要用户输入姓名，但用户却输入了一个 SQLite 语句，而这语句就会在不知不觉中在数据库上运行。

永远不要相信用户提供的数据，所以只处理通过验证的数据，这项规则是通过模式匹配来完成的。在下面的实例中，用户名 username 被限制为字母数字字符或者下划线，长度必须在 8 到 20 个字符之间 - 请根据需要修改这些规则。

```
if (preg_match("/^\w{8,20}$/", $_GET['username'], $matches)){
   $db = new SQLiteDatabase('filename');
   $result = @$db->query("SELECT * FROM users WHERE username=$match
}else{
   echo "username not accepted";
}
```

为了演示这个问题，假设考虑此摘录：To demonstrate the problem, consider this excerpt:

```
$name = "Qadir'; DELETE FROM users;";
@$db->query("SELECT * FROM users WHERE username='{$name}'");
```

函数调用是为了从用户表中检索 name 列与用户指定的名称相匹配的记录。正常情况下，**$name** 只包含字母数字字符或者空格，比如字符串 ilia。但在这里，向 $name 追加了一个全新的查询，这个对数据库的调用将会造成灾难性的问题：注入的 DELETE 查询会删除 users 的所有记录。

虽然已经存在有不允许查询堆叠或在单个函数调用中执行多个查询的数据库接口，如果尝试堆叠查询，则会调用失败，但 SQLite 和 PostgreSQL 里仍进行堆叠查询，即执行在一个字符串中提供的所有查询，这会导致严重的安全问题。

防止 SQL 注入

在脚本语言中，比如 PERL 和 PHP，您可以巧妙地处理所有的转义字符。编程语言 PHP 提供了字符串函数 **sqlite_escape_string()** 来转义对于 SQLite 来说比较特殊的输入字符。

```
if (get_magic_quotes_gpc())
{
  $name = sqlite_escape_string($name);
}
$result = @$db->query("SELECT * FROM users WHERE username='{$name}
```

虽然编码使得插入数据变得安全，但是它会呈现简单的文本比较，在查询中，对于包含二进制数据的列，**LIKE** 子句是不可用的。

请注意，addslashes() 不应该被用在 SQLite 查询中引用字符串，它会在检索数据时导致奇怪的结果。

SQLite Explain（解释）

在 SQLite 语句之前，可以使用 "EXPLAIN" 关键字或 "EXPLAIN QUERY PLAN" 短语，用于描述表的细节。

如果省略了 EXPLAIN 关键字或短语，任何的修改都会引起 SQLite 语句的查询行为，并返回有关 SQLite 语句如何操作的信息。

- 来自 EXPLAIN 和 EXPLAIN QUERY PLAN 的输出只用于交互式分析和排除故障。
- 输出格式的细节可能会随着 SQLite 版本的不同而有所变化。
- 应用程序不应该使用 EXPLAIN 或 EXPLAIN QUERY PLAN，因为其确切的行为是可变的且只有部分会被记录。

语法

EXPLAIN 的语法如下：

```
EXPLAIN [SQLite Query]
```

EXPLAIN QUERY PLAN 的语法如下：

```
EXPLAIN  QUERY PLAN [SQLite Query]
```

实例

假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

现在，让我们检查 SELECT 语句中的 **Explain** 使用：

```
sqlite> EXPLAIN SELECT *  FROM COMPANY  WHERE Salary &gt;= 20000;
```

这将产生以下结果：

```
addr        opcode      p1          p2          p3
----------  ----------  ----------  ----------  ----------
0           Goto        0           19
1           Integer     0           0
2           OpenRead    0           8
3           SetNumColu  0           5
4           Rewind      0           17
5           Column      0           4
6           RealAffini  0           0
7           Integer     20000       0
8           Lt          357         16          collseq(BI
9           Rowid       0           0
10          Column      0           1
11          Column      0           2
12          Column      0           3
13          Column      0           4
14          RealAffini  0           0
15          Callback    5           0
16          Next        0           5
17          Close       0           0
18          Halt        0           0
19          Transactio  0           0
20          VerifyCook  0           38
21          Goto        0           1
22          Noop        0           0
```

现在，让我们检查 SELECT 语句中的 **Explain Query Plan** 使用：

```
SQLite> EXPLAIN QUERY PLAN SELECT * FROM COMPANY WHERE Salary &gt;=
```

```
order       from        detail
----------  ----------  -------------
0           0           TABLE COMPANY
```

SQLite Vacuum

VACUUM 命令通过复制主数据库中的内容到一个临时数据库文件，然后清空主数据库，并从副本中重新载入原始的数据库文件。这消除了空闲页，把表中的数据排列为连续的，另外会清理数据库文件结构。

如果表中没有明确的整型主键（INTEGER PRIMARY KEY），VACUUM 命令可能会改变表中条目的行 ID（ROWID）。VACUUM 命令只适用于主数据库，附加的数据库文件是不可能使用 VACUUM 命令。

如果有一个活动的事务，VACUUM 命令就会失败。VACUUM 命令是一个用于内存数据库的任何操作。由于 VACUUM 命令从头开始重新创建数据库文件，所以 VACUUM 也可以用于修改许多数据库特定的配置参数。

手动 VACUUM

下面是在命令提示符中对整个数据库发出 VACUUM 命令的语法：

```
$sqlite3 database_name "VACUUM;"
```

您也可以在 SQLite 提示符中运行 VACUUM，如下所示：

```
sqlite> VACUUM;
```

您也可以在特定的表上运行 VACUUM，如下所示：

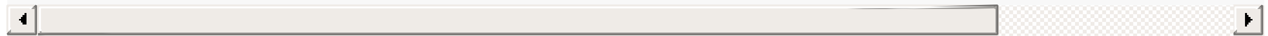
```
sqlite> VACUUM table_name;
```

自动 VACCUM（Auto-VACUUM）

SQLite 的 Auto-VACUUM 与 VACUUM 不大一样，它只是把空闲页移到数据库末尾，从而减小数据库大小。通过这样做，它可以明显地把数据库碎片化，而 VACUUM 则是反碎片化。所以 Auto-VACUUM 只会让数据库更小。

在 SQLite 提示符中，您可以通过下面的编译运行，启用/禁用 SQLite 的 Auto-VACUUM：

```
sqlite> PRAGMA auto_vacuum = NONE;  -- 0 means disable auto vacuum
sqlite> PRAGMA auto_vacuum = INCREMENTAL;  -- 1 means enable incren
sqlite> PRAGMA auto_vacuum = FULL;  -- 2 means enable full auto vac
```



您可以从命令提示符中运行下面的命令来检查 auto-vacuum 设置：

```
$sqlite3 database_name "PRAGMA auto_vacuum;"
```

SQLite 日期 & 时间

SQLite 支持以下五个日期和时间函数：

函数	实例
date(timestring, modifiers...)	以 YYYY-MM-DD 格式返回日期。
time(timestring, modifiers...)	以 HH:MM:SS 格式返回时间。
datetime(timestring, modifiers...)	以 YYYY-MM-DD HH:MM:SS 格式返回。
julianday(timestring, modifiers...)	这将返回从格林尼治时间的公元前 4714 年 11 月 24 日正午算起的天数。
strftime(timestring, modifiers...)	这将根据第一个参数指定的格式字符串返回格式化的日期。具体格式见下边讲解。

上述五个日期和时间函数把时间字符串作为参数。时间字符串后跟零个或多个 modifiers 修饰符。strftime() 函数也可以把格式字符串作为其第一个参数。下面将为您详细讲解不同类型的时间字符串和修饰符。

时间字符串

一个时间字符串可以采用下面任何一种格式：

时间字符串	实例
YYYY-MM-DD	2010-12-30
YYYY-MM-DD HH:MM	2010-12-30 12:10
YYYY-MM-DD HH:MM:SS.SSS	2010-12-30 12:10:04.100
MM-DD-YYYY HH:MM	30-12-2010 12:10
HH:MM	12:10
YYYY-MM-DD**T**HH:MM	2010-12-30 12:10
HH:MM:SS	12:10:01
YYYYMMDD HHMMSS	20101230 121001
now	2013-05-07

您可以使用 "T" 作为分隔日期和时间的文字字符。

修饰符（Modifiers）

时间字符串后边可跟着零个或多个的修饰符，这将改变有上述五个函数返回的日期和/或时间。任何上述五大功能返回时间。修饰符应从左到右使用，下面列出了可在 SQLite 中使用的修饰符：

- NNN days
- NNN hours
- NNN minutes
- NNN.NNNN seconds
- NNN months
- NNN years
- start of month
- start of year
- start of day
- weekday N
- unixepoch
- localtime
- utc

格式化

SQLite 提供了非常方便的函数 **strftime()** 来格式化任何日期和时间。您可以使用以下的替换来格式化日期和时间：

替换	描述
%d	一月中的第几天，01-31
%f	带小数部分的秒，SS.SSS
%H	小时，00-23
%j	一年中的第几天，001-366
%J	儒略日数，DDDD.DDDD
%m	月，00-12
%M	分，00-59
%s	从 1970-01-01 算起的秒数
%S	秒，00-59
%w	一周中的第几天，0-6 (0 is Sunday)
%W	一年中的第几周，01-53
%Y	年，YYYY
%%	% symbol

实例

现在让我们使用 SQLite 提示符尝试不同的实例。下面是计算当前日期：

```
sqlite> SELECT date('now');
2013-05-07
```

下面是计算当前月份的最后一天：

```
sqlite> SELECT date('now','start of month','+1 month','-1 day');
2013-05-31
```

下面是计算给定 UNIX 时间戳 1092941466 的日期和时间：

```
sqlite> SELECT datetime(1092941466, 'unixepoch');
2004-08-19 18:51:06
```

下面是计算给定 UNIX 时间戳 1092941466 相对本地时区的日期和时间：

```
sqlite> SELECT datetime(1092941466, 'unixepoch', 'localtime');
2004-08-19 11:51:06
```

下面是计算当前的 UNIX 时间戳：

```
sqlite> SELECT datetime(1092941466, 'unixepoch', 'localtime');
1367926057
```

下面是计算美国"独立宣言"签署以来的天数：

```
sqlite> SELECT julianday('now') - julianday('1776-07-04');
86504.4775830326
```

下面是计算从 2004 年某一特定时刻以来的秒数：

```
sqlite> SELECT strftime('%s','now') - strftime('%s','2004-01-01 02
295001572
```

下面是计算当年 10 月的第一个星期二的日期：

```
sqlite> SELECT date('now','start of year','+9 months','weekday 2')
2013-10-01
```

下面是计算从 UNIX 纪元算起的以秒为单位的时间（类似 strftime('%s','now')，不同的是这里有包括小数部分）：

```
sqlite> SELECT (julianday('now') - 2440587.5)*86400.0;
1367926077.12598
```

在 UTC 与本地时间值之间进行转换，当格式化日期时，使用 utc 或 localtime 修饰符，如下所示：

```
sqlite> SELECT time('12:00', 'localtime');
05:00:00
```

```
sqlite>  SELECT time('12:00', 'utc');
19:00:00
```

SQLite 常用函数

SQLite 有许多内置函数用于处理字符串或数字数据。下面列出了一些有用的 SQLite 内置函数，且所有函数都是大小写不敏感，这意味着您可以使用这些函数的小写形式或大写形式或混合形式。欲了解更多详情，请查看 SQLite 的官方文档：

函数	描述
SQLite COUNT 函数	SQLite COUNT 聚集函数是用来计算一个数据库表中的行数。
SQLite MAX 函数	SQLite MAX 聚合函数允许我们选择某列的最大值。
SQLite MIN 函数	SQLite MIN 聚合函数允许我们选择某列的最小值。
SQLite AVG 函数	SQLite AVG 聚合函数计算某列的平均值。
SQLite SUM 函数	SQLite SUM 聚合函数允许为一个数值列计算总和。
SQLite RANDOM 函数	SQLite RANDOM 函数返回一个介于 -9223372036854775808 和 +9223372036854775807 之间的伪随机整数。
SQLite ABS 函数	SQLite ABS 函数返回数值参数的绝对值。
SQLite UPPER 函数	SQLite UPPER 函数把字符串转换为大写字母。
SQLite LOWER 函数	SQLite LOWER 函数把字符串转换为小写字母。
SQLite LENGTH 函数	SQLite LENGTH 函数返回字符串的长度。
SQLite sqlite_version 函数	SQLite sqlite_version 函数返回 SQLite 库的版本。

在我们开始讲解这些函数实例之前，先假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

SQLite COUNT 函数

SQLite COUNT 聚集函数是用来计算一个数据库表中的行数。下面是实例：

```
sqlite> SELECT count(*) FROM COMPANY;
```

上面的 SQLite SQL 语句将产生以下结果：

```
count(*)
----------
7
```

SQLite MAX 函数

SQLite MAX 聚合函数允许我们选择某列的最大值。下面是实例：

```
sqlite> SELECT max(salary) FROM COMPANY;
```

上面的 SQLite SQL 语句将产生以下结果：

```
max(salary)
-----------
85000.0
```

SQLite MIN 函数

SQLite MIN 聚合函数允许我们选择某列的最小值。下面是实例：

```
sqlite> SELECT min(salary) FROM COMPANY;
```

上面的 SQLite SQL 语句将产生以下结果：

```
min(salary)
-----------
10000.0
```

SQLite AVG 函数

SQLite AVG 聚合函数计算某列的平均值。下面是实例：

```
sqlite> SELECT avg(salary) FROM COMPANY;
```

上面的 SQLite SQL 语句将产生以下结果：

```
avg(salary)
----------------
37142.8571428572
```

SQLite SUM 函数

SQLite SUM 聚合函数允许为一个数值列计算总和。下面是实例：

```
sqlite> SELECT sum(salary) FROM COMPANY;
```

上面的 SQLite SQL 语句将产生以下结果：

```
sum(salary)
-----------
260000.0
```

SQLite RANDOM 函数

SQLite RANDOM 函数返回一个介于 -9223372036854775808 和 +9223372036854775807 之间的伪随机整数。下面是实例：

```
sqlite> SELECT random() AS Random;
```

上面的 SQLite SQL 语句将产生以下结果：

```
Random
--------------------
5876796417670984050
```

SQLite ABS 函数

SQLite ABS 函数返回数值参数的绝对值。下面是实例：

```
sqlite> SELECT abs(5), abs(-15), abs(NULL), abs(0), abs("ABC");
```

上面的 SQLite SQL 语句将产生以下结果：

```
abs(5)      abs(-15)    abs(NULL)   abs(0)      abs("ABC")
----------  ----------  ----------  ----------  ----------
5           15                      0           0.0
```

SQLite UPPER 函数

SQLite UPPER 函数把字符串转换为大写字母。下面是实例：

```
sqlite> SELECT upper(name) FROM COMPANY;
```

上面的 SQLite SQL 语句将产生以下结果：

```
upper(name)
-----------
PAUL
ALLEN
TEDDY
MARK
DAVID
KIM
JAMES
```

SQLite LOWER 函数

SQLite LOWER 函数把字符串转换为小写字母。下面是实例：

```
sqlite> SELECT lower(name) FROM COMPANY;
```

上面的 SQLite SQL 语句将产生以下结果：

```
lower(name)
-----------
paul
allen
teddy
mark
david
kim
james
```

SQLite LENGTH 函数

SQLite LENGTH 函数返回字符串的长度。下面是实例：

```
sqlite> SELECT name, length(name) FROM COMPANY;
```

上面的 SQLite SQL 语句将产生以下结果：

```
NAME        length(name)
----------  ------------
Paul        4
Allen       5
Teddy       5
Mark        4
David       5
Kim         3
James       5
```

SQLite sqlite_version 函数

SQLite sqlite_version 函数返回 SQLite 库的版本。下面是实例：

```
sqlite> SELECT sqlite_version() AS 'SQLite Version';
```

上面的 SQLite SQL 语句将产生以下结果：

```
SQLite Version
--------------
3.6.20
```

SQLite 接口

SQLite - C/C++

安装

在 C/C++ 程序中使用 SQLite 之前，我们需要确保机器上已经有 SQLite 库。可以查看 SQLite 安装章节了解安装过程。

C/C++ 接口 API

以下是重要的 C&C++ / SQLite 接口程序，可以满足您在 C/C++ 程序中使用 SQLite 数据库的需求。如果您需要了解更多细节，请查看 SQLite 官方文档。

API	描述
sqlite3_open(const char *filename, sqlite3 **ppDb)	该例程打开一个指向 SQLite 数据库文件的连接，返回一个用于其他 SQLite 程序的数据库连接对象。如果 *filename* 参数是 NULL 或 ':memory:'，那么 sqlite3_open() 将会在 RAM 中创建一个内存数据库，这只会在 session 的有效时间内持续。 如果文件名 filename 不为 NULL，那么 sqlite3_open() 将使用这个参数值尝试打开数据库文件。如果该名称的文件不存在，sqlite3_open() 将创建一个新的命名为该名称的数据库文件并打开。
sqlite3_exec(sqlite3*, const char *sql, sqlite_callback, void *data, char **errmsg)	该例程提供了一个执行 SQL 命令的快捷方式，SQL 命令由 sql 参数提供，可以由多个 SQL 命令组成。在这里，第一个参数 *sqlite3* 是打开的数据库对象，*sqlite_callback* 是一个回调，*data* 作为其第一个参数，errmsg 将被返回用来获取程序生成的任何错误。 sqlite3_exec() 程序解析并执行由 **sql** 参数所给的每个命令，直到字符串结束或者遇到错误为止。
sqlite3_close(sqlite3*)	该例程关闭之前调用 sqlite3_open() 打开的数据库连接。所有与连接相关的语句都应在连接关闭之前完成。 如果还有查询没有完成，sqlite3_close() 将返回 SQLITE_BUSY 禁止关闭的错误消息。

连接数据库

下面的 C 代码段显示了如何连接到一个现有的数据库。如果数据库不存在，那么它就会被创建，最后将返回一个数据库对象。

```
#include <stdio.h>
#include <sqlite3.h>

int main(int argc, char* argv[])
{
   sqlite3 *db;
   char *zErrMsg = 0;
   int rc;

   rc = sqlite3_open("test.db", &db);

   if( rc ){
      fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(d
      exit(0);
   }else{
      fprintf(stderr, "Opened database successfully\n");
   }
   sqlite3_close(db);
}
```

现在，让我们来编译和运行上面的程序，在当前目录中创建我们的数据库 **test.db**。您可以根据需要改变路径。

```
$gcc test.c -l sqlite3
$./a.out
Opened database successfully
```

如果要使用 C++ 源代码，可以按照下列所示编译代码：

```
$g++ test.c -l sqlite3
```

在这里，把我们的程序链接上 sqlite3 库，以便向 C 程序提供必要的函数。这将在您的目录下创建一个数据库文件 test.db，您将得到如下结果：

```
-rwxr-xr-x. 1 root root 7383 May  8 02:06 a.out
-rw-r--r--. 1 root root  323 May  8 02:05 test.c
-rw-r--r--. 1 root root    0 May  8 02:06 test.db
```

创建表

下面的 C 代码段将用于在先前创建的数据库中创建一个表：

```
#include <stdio.h>
#include <stdlib.h>
#include <sqlite3.h>

static int callback(void *NotUsed, int argc, char **argv, char **az
   int i;
   for(i=0; i&lt;argc; i++){
      printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL")
   }
   printf("\n");
   return 0;
}

int main(int argc, char* argv[])
{
   sqlite3 *db;
   char *zErrMsg = 0;
   int  rc;
   char *sql;

   /* Open database */
   rc = sqlite3_open("test.db", &db);
   if( rc ){
      fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(d
      exit(0);
   }else{
      fprintf(stdout, "Opened database successfully\n");
   }

   /* Create SQL statement */
   sql = "CREATE TABLE COMPANY("  \
         "ID INT PRIMARY KEY     NOT NULL," \
         "NAME           TEXT    NOT NULL," \
         "AGE            INT     NOT NULL," \
         "ADDRESS        CHAR(50)," \
         "SALARY         REAL );";

   /* Execute SQL statement */
   rc = sqlite3_exec(db, sql, callback, 0, &zErrMsg);
   if( rc != SQLITE_OK ){
   fprintf(stderr, "SQL error: %s\n", zErrMsg);
      sqlite3_free(zErrMsg);
   }else{
      fprintf(stdout, "Table created successfully\n");
   }
   sqlite3_close(db);
   return 0;
}
```

上述程序编译和执行时，它会在 test.db 文件中创建 COMPANY 表，最终文件列表如下所示：

```
-rwxr-xr-x. 1 root root 9567 May  8 02:31 a.out
-rw-r--r--. 1 root root 1207 May  8 02:31 test.c
-rw-r--r--. 1 root root 3072 May  8 02:31 test.db
```

INSERT 操作

下面的 C 代码段显示了如何在上面创建的 COMPANY 表中创建记录：

```
#include <stdio.h>
#include <stdlib.h>
#include <sqlite3.h>

static int callback(void *NotUsed, int argc, char **argv, char **az
   int i;
   for(i=0; i&lt;argc; i++){
      printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL")
   }
   printf("\n");
   return 0;
}

int main(int argc, char* argv[])
{
   sqlite3 *db;
   char *zErrMsg = 0;
   int rc;
   char *sql;

   /* Open database */
   rc = sqlite3_open("test.db", &db);
   if( rc ){
      fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(c
      exit(0);
   }else{
      fprintf(stderr, "Opened database successfully\n");
   }

   /* Create SQL statement */
   sql = "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) "  \
         "VALUES (1, 'Paul', 32, 'California', 20000.00 ); " \
         "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) "  \
         "VALUES (2, 'Allen', 25, 'Texas', 15000.00 ); "     \
         "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)" \
         "VALUES (3, 'Teddy', 23, 'Norway', 20000.00 );" \
         "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)" \
         "VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 );";

   /* Execute SQL statement */
   rc = sqlite3_exec(db, sql, callback, 0, &zErrMsg);
   if( rc != SQLITE_OK ){
      fprintf(stderr, "SQL error: %s\n", zErrMsg);
      sqlite3_free(zErrMsg);
   }else{
      fprintf(stdout, "Records created successfully\n");
   }
   sqlite3_close(db);
   return 0;
}
```

上述程序编译和执行时，它会在 COMPANY 表中创建给定记录，并会显示以下两行：

```
Opened database successfully
Records created successfully
```

SELECT 操作

在我们开始讲解获取记录的实例之前，让我们先了解下回调函数的一些细节，这将在我们的实例使用到。这个回调提供了一个从 SELECT 语句获得结果的方式。它声明如下：

```
typedef int (*sqlite3_callback)(
void*,    /* Data provided in the 4th argument of sqlite3_exec() */
int,      /* The number of columns in row */
char**,   /* An array of strings representing fields in the row */
char**    /* An array of strings representing column names */
);
```

如果上面的回调在 sqlite_exec() 程序中作为第三个参数，那么 SQLite 将为 SQL 参数内执行的每个 SELECT 语句中处理的每个记录调用这个回调函数。

下面的 C 代码段显示了如何从前面创建的 COMPANY 表中获取并显示记录：

```
#include <stdio.h>
#include <stdlib.h>
#include <sqlite3.h>

static int callback(void *data, int argc, char **argv, char **azCol
   int i;
   fprintf(stderr, "%s: ", (const char*)data);
   for(i=0; i&lt;argc; i++){
      printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL")
   }
   printf("\n");
   return 0;
}

int main(int argc, char* argv[])
{
   sqlite3 *db;
   char *zErrMsg = 0;
   int rc;
   char *sql;
   const char* data = "Callback function called";

   /* Open database */
   rc = sqlite3_open("test.db", &db);
   if( rc ){
      fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(
      exit(0);
   }else{
      fprintf(stderr, "Opened database successfully\n");
   }

   /* Create SQL statement */
   sql = "SELECT * from COMPANY";

   /* Execute SQL statement */
   rc = sqlite3_exec(db, sql, callback, (void*)data, &zErrMsg);
   if( rc != SQLITE_OK ){
      fprintf(stderr, "SQL error: %s\n", zErrMsg);
      sqlite3_free(zErrMsg);
   }else{
      fprintf(stdout, "Operation done successfully\n");
   }
   sqlite3_close(db);
   return 0;
}
```

上述程序编译和执行时，它会产生以下结果：

```
Opened database successfully
Callback function called: ID = 1
NAME = Paul
AGE = 32
ADDRESS = California
SALARY = 20000.0

Callback function called: ID = 2
NAME = Allen
AGE = 25
ADDRESS = Texas
SALARY = 15000.0

Callback function called: ID = 3
NAME = Teddy
AGE = 23
ADDRESS = Norway
SALARY = 20000.0

Callback function called: ID = 4
NAME = Mark
AGE = 25
ADDRESS = Rich-Mond
SALARY = 65000.0

Operation done successfully
```

UPDATE 操作

下面的 C 代码段显示了如何使用 UPDATE 语句来更新任何记录，然后从 COMPANY 表中获取并显示更新的记录：

```
#include <stdio.h>
#include <stdlib.h>
#include <sqlite3.h>

static int callback(void *data, int argc, char **argv, char **azCol
   int i;
   fprintf(stderr, "%s: ", (const char*)data);
   for(i=0; i&lt;argc; i++){
      printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL")
   }
   printf("\n");
   return 0;
}

int main(int argc, char* argv[])
{
   sqlite3 *db;
   char *zErrMsg = 0;
   int rc;
   char *sql;
   const char* data = "Callback function called";

   /* Open database */
   rc = sqlite3_open("test.db", &db);
   if( rc ){
      fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(d
      exit(0);
   }else{
      fprintf(stderr, "Opened database successfully\n");
   }

   /* Create merged SQL statement */
   sql = "UPDATE COMPANY set SALARY = 25000.00 where ID=1; " \
         "SELECT * from COMPANY";

   /* Execute SQL statement */
   rc = sqlite3_exec(db, sql, callback, (void*)data, &zErrMsg);
   if( rc != SQLITE_OK ){
      fprintf(stderr, "SQL error: %s\n", zErrMsg);
      sqlite3_free(zErrMsg);
   }else{
      fprintf(stdout, "Operation done successfully\n");
   }
   sqlite3_close(db);
   return 0;
}
```

上述程序编译和执行时，它会产生以下结果：

```
Opened database successfully
Callback function called: ID = 1
NAME = Paul
AGE = 32
ADDRESS = California
SALARY = 25000.0

Callback function called: ID = 2
NAME = Allen
AGE = 25
ADDRESS = Texas
SALARY = 15000.0

Callback function called: ID = 3
NAME = Teddy
AGE = 23
ADDRESS = Norway
SALARY = 20000.0

Callback function called: ID = 4
NAME = Mark
AGE = 25
ADDRESS = Rich-Mond
SALARY = 65000.0

Operation done successfully
```

DELETE 操作

下面的 C 代码段显示了如何使用 DELETE 语句删除任何记录，然后从 COMPANY 表中获取并显示剩余的记录：

```
#include <stdio.h>
#include <stdlib.h>
#include <sqlite3.h>

static int callback(void *data, int argc, char **argv, char **azCol
   int i;
   fprintf(stderr, "%s: ", (const char*)data);
   for(i=0; i<argc; i++){
      printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL")
   }
   printf("\n");
   return 0;
}

int main(int argc, char* argv[])
{
   sqlite3 *db;
   char *zErrMsg = 0;
   int rc;
   char *sql;
   const char* data = "Callback function called";

   /* Open database */
   rc = sqlite3_open("test.db", &db);
   if( rc ){
      fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(d
      exit(0);
   }else{
      fprintf(stderr, "Opened database successfully\n");
   }

   /* Create merged SQL statement */
   sql = "DELETE from COMPANY where ID=2; " \
         "SELECT * from COMPANY";

   /* Execute SQL statement */
   rc = sqlite3_exec(db, sql, callback, (void*)data, &zErrMsg);
   if( rc != SQLITE_OK ){
      fprintf(stderr, "SQL error: %s\n", zErrMsg);
      sqlite3_free(zErrMsg);
   }else{
      fprintf(stdout, "Operation done successfully\n");
   }
   sqlite3_close(db);
   return 0;
}
```

上述程序编译和执行时，它会产生以下结果：

```
Opened database successfully
Callback function called: ID = 1
NAME = Paul
AGE = 32
ADDRESS = California
SALARY = 20000.0

Callback function called: ID = 3
NAME = Teddy
AGE = 23
ADDRESS = Norway
SALARY = 20000.0

Callback function called: ID = 4
NAME = Mark
AGE = 25
ADDRESS = Rich-Mond
SALARY = 65000.0

Operation done successfully
```

SQLite - Java

安装

在 Java 程序中使用 SQLite 之前，我们需要确保机器上已经有 SQLite JDBC Driver 驱动程序和 Java。可以查看 Java 教程了解如何在计算机上安装 Java。现在，我们来看看如何在机器上安装 SQLite JDBC 驱动程序。

- 从 sqlite-jdbc 库下载 *sqlite-jdbc-(VERSION).jar* 的最新版本。
- 在您的 class 路径中添加下载的 jar 文件 *sqlite-jdbc-(VERSION).jar*，或者在 -classpath 选项中使用它，这将在后面的实例中进行讲解。

在学习下面部分的知识之前，您必须对 Java JDBC 概念有初步了解。如果您还未了解相关知识，那么建议您可以先花半个小时学习下 JDBC 教程相关知识，这将有助于您学习接下来讲解的知识。

连接数据库

下面的 Java 程序显示了如何连接到一个现有的数据库。如果数据库不存在，那么它就会被创建，最后将返回一个数据库对象。

```
import java.sql.*;

public class SQLiteJDBC
{
  public static void main( String args[] )
  {
    Connection c = null;
    try {
      Class.forName("org.sqlite.JDBC");
      c = DriverManager.getConnection("jdbc:sqlite:test.db");
    } catch ( Exception e ) {
      System.err.println( e.getClass().getName() + ": " + e.getMess
      System.exit(0);
    }
    System.out.println("Opened database successfully");
  }
}
```

现在，让我们来编译和运行上面的程序，在当前目录中创建我们的数据库 **test.db**。您可以根据需要改变路径。我们假设当前路径下可用的 JDBC 驱动程序的版本是 *sqlite-jdbc-3.7.2.jar*。

```
$javac SQLiteJDBC.java
$java -classpath ".:sqlite-jdbc-3.7.2.jar" SQLiteJDBC
Open database successfully
```

如果您想要使用 Windows 机器，可以按照下列所示编译和运行您的代码：

```
$javac SQLiteJDBC.java
$java -classpath ".;sqlite-jdbc-3.7.2.jar" SQLiteJDBC
Opened database successfully
```

创建表

下面的 Java 程序将用于在先前创建的数据库中创建一个表：

```
import java.sql.*;

public class SQLiteJDBC
{
  public static void main( String args[] )
  {
    Connection c = null;
    Statement stmt = null;
    try {
      Class.forName("org.sqlite.JDBC");
      c = DriverManager.getConnection("jdbc:sqlite:test.db");
      System.out.println("Opened database successfully");

      stmt = c.createStatement();
      String sql = "CREATE TABLE COMPANY " +
                   "(ID INT PRIMARY KEY     NOT NULL," +
                   " NAME           TEXT    NOT NULL, " +
                   " AGE            INT     NOT NULL, " +
                   " ADDRESS        CHAR(50), " +
                   " SALARY         REAL)";
      stmt.executeUpdate(sql);
      stmt.close();
      c.close();
    } catch ( Exception e ) {
      System.err.println( e.getClass().getName() + ": " + e.getMess
      System.exit(0);
    }
    System.out.println("Table created successfully");
  }
}
```

上述程序编译和执行时，它会在 **test.db** 中创建 COMPANY 表，最终文件列表如下所示：

```
-rw-r--r--. 1 root root 3201128 Jan 22 19:04 sqlite-jdbc-3.7.2.jar
-rw-r--r--. 1 root root    1506 May  8 05:43 SQLiteJDBC.class
-rw-r--r--. 1 root root     832 May  8 05:42 SQLiteJDBC.java
-rw-r--r--. 1 root root    3072 May  8 05:43 test.db
```

INSERT 操作

下面的 Java 代码显示了如何在上面创建的 COMPANY 表中创建记录：

```
import java.sql.*;

public class SQLiteJDBC
{
  public static void main( String args[] )
  {
    Connection c = null;
    Statement stmt = null;
    try {
      Class.forName("org.sqlite.JDBC");
      c = DriverManager.getConnection("jdbc:sqlite:test.db");
      c.setAutoCommit(false);
      System.out.println("Opened database successfully");

      stmt = c.createStatement();
      String sql = "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY
                   "VALUES (1, 'Paul', 32, 'California', 20000.00 
      stmt.executeUpdate(sql);

      sql = "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) " +
            "VALUES (2, 'Allen', 25, 'Texas', 15000.00 );";
      stmt.executeUpdate(sql);

      sql = "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) " +
            "VALUES (3, 'Teddy', 23, 'Norway', 20000.00 );";
      stmt.executeUpdate(sql);

      sql = "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) " +
            "VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 );";
      stmt.executeUpdate(sql);

      stmt.close();
      c.commit();
      c.close();
    } catch ( Exception e ) {
      System.err.println( e.getClass().getName() + ": " + e.getMess
      System.exit(0);
    }
    System.out.println("Records created successfully");
  }
}
```

上述程序编译和执行时，它会在 COMPANY 表中创建给定记录，并会显示以下两行：

```
Opened database successfully
Records created successfully
```

SELECT 操作

下面的 Java 程序显示了如何从前面创建的 COMPANY 表中获取并显示记录：

```
import java.sql.*;

public class SQLiteJDBC
{
  public static void main( String args[] )
  {
    Connection c = null;
    Statement stmt = null;
    try {
      Class.forName("org.sqlite.JDBC");
      c = DriverManager.getConnection("jdbc:sqlite:test.db");
      c.setAutoCommit(false);
      System.out.println("Opened database successfully");

      stmt = c.createStatement();
      ResultSet rs = stmt.executeQuery( "SELECT * FROM COMPANY;" );
      while ( rs.next() ) {
         int id = rs.getInt("id");
         String  name = rs.getString("name");
         int age  = rs.getInt("age");
         String  address = rs.getString("address");
         float salary = rs.getFloat("salary");
         System.out.println( "ID = " + id );
         System.out.println( "NAME = " + name );
         System.out.println( "AGE = " + age );
         System.out.println( "ADDRESS = " + address );
         System.out.println( "SALARY = " + salary );
         System.out.println();
      }
      rs.close();
      stmt.close();
      c.close();
    } catch ( Exception e ) {
      System.err.println( e.getClass().getName() + ": " + e.getMess
      System.exit(0);
    }
    System.out.println("Operation done successfully");
  }
}
```

上述程序编译和执行时，它会产生以下结果：

```
Opened database successfully
ID = 1
NAME = Paul
AGE = 32
ADDRESS = California
SALARY = 20000.0

ID = 2
NAME = Allen
AGE = 25
ADDRESS = Texas
SALARY = 15000.0

ID = 3
NAME = Teddy
AGE = 23
ADDRESS = Norway
SALARY = 20000.0

ID = 4
NAME = Mark
AGE = 25
ADDRESS = Rich-Mond
SALARY = 65000.0

Operation done successfully
```

UPDATE 操作

下面的 Java 代码显示了如何使用 UPDATE 语句来更新任何记录，然后从 COMPANY 表中获取并显示更新的记录：

```
import java.sql.*;

public class SQLiteJDBC
{
  public static void main( String args[] )
  {
    Connection c = null;
    Statement stmt = null;
    try {
      Class.forName("org.sqlite.JDBC");
      c = DriverManager.getConnection("jdbc:sqlite:test.db");
      c.setAutoCommit(false);
      System.out.println("Opened database successfully");

      stmt = c.createStatement();
      String sql = "UPDATE COMPANY set SALARY = 25000.00 where ID=1
      stmt.executeUpdate(sql);
      c.commit();

      ResultSet rs = stmt.executeQuery( "SELECT * FROM COMPANY;" );
      while ( rs.next() ) {
         int id = rs.getInt("id");
         String  name = rs.getString("name");
         int age  = rs.getInt("age");
         String  address = rs.getString("address");
         float salary = rs.getFloat("salary");
         System.out.println( "ID = " + id );
         System.out.println( "NAME = " + name );
         System.out.println( "AGE = " + age );
         System.out.println( "ADDRESS = " + address );
         System.out.println( "SALARY = " + salary );
         System.out.println();
      }
      rs.close();
      stmt.close();
      c.close();
    } catch ( Exception e ) {
      System.err.println( e.getClass().getName() + ": " + e.getMess
      System.exit(0);
    }
    System.out.println("Operation done successfully");
  }
}
```

上述程序编译和执行时，它会产生以下结果：

```
Opened database successfully
ID = 1
NAME = Paul
AGE = 32
ADDRESS = California
SALARY = 25000.0

ID = 2
NAME = Allen
AGE = 25
ADDRESS = Texas
SALARY = 15000.0

ID = 3
NAME = Teddy
AGE = 23
ADDRESS = Norway
SALARY = 20000.0

ID = 4
NAME = Mark
AGE = 25
ADDRESS = Rich-Mond
SALARY = 65000.0

Operation done successfully
```

DELETE 操作

下面的 Java 代码显示了如何使用 DELETE 语句删除任何记录，然后从 COMPANY 表中获取并显示剩余的记录：

```
import java.sql.*;

public class SQLiteJDBC
{
  public static void main( String args[] )
  {
    Connection c = null;
    Statement stmt = null;
    try {
      Class.forName("org.sqlite.JDBC");
      c = DriverManager.getConnection("jdbc:sqlite:test.db");
      c.setAutoCommit(false);
      System.out.println("Opened database successfully");

      stmt = c.createStatement();
      String sql = "DELETE from COMPANY where ID=2;";
      stmt.executeUpdate(sql);
      c.commit();

      ResultSet rs = stmt.executeQuery( "SELECT * FROM COMPANY;" );
      while ( rs.next() ) {
         int id = rs.getInt("id");
         String  name = rs.getString("name");
         int age  = rs.getInt("age");
         String  address = rs.getString("address");
         float salary = rs.getFloat("salary");
         System.out.println( "ID = " + id );
         System.out.println( "NAME = " + name );
         System.out.println( "AGE = " + age );
         System.out.println( "ADDRESS = " + address );
         System.out.println( "SALARY = " + salary );
         System.out.println();
      }
      rs.close();
      stmt.close();
      c.close();
    } catch ( Exception e ) {
      System.err.println( e.getClass().getName() + ": " + e.getMess
      System.exit(0);
    }
    System.out.println("Operation done successfully");
  }
}
```

上述程序编译和执行时，它会产生以下结果：

```
Opened database successfully
ID = 1
NAME = Paul
AGE = 32
ADDRESS = California
SALARY = 25000.0

ID = 3
NAME = Teddy
AGE = 23
ADDRESS = Norway
SALARY = 20000.0

ID = 4
NAME = Mark
AGE = 25
ADDRESS = Rich-Mond
SALARY = 65000.0

Operation done successfully
```

SQLite - PHP

安装

自 PHP 5.3.0 起默认启用 SQLite3 扩展。可以在编译时使用 **--without-sqlite3** 禁用 SQLite3 扩展。

Windows 用户必须启用 php_sqlite3.dll 才能使用该扩展。自 PHP 5.3.0 起，这个 DLL 被包含在 PHP 的 Windows 分发版中。

如需了解详细的安装指导，建议查看我们的 PHP 教程和它的官方网站。

PHP 接口 API

以下是重要的 PHP 程序，可以满足您在 PHP 程序中使用 SQLite 数据库的需求。如果您需要了解更多细节，请查看 PHP 官方文档。

API	描述
public void SQLite3::open (filename, flags, encryption_key)	打开一个 SQLite 3 数据库。如果构建包括加密，那么它将尝试使用的密钥。 如果文件名 *filename* 赋值为 **':memory:'**，那么 SQLite3::open() 将会在 RAM 中创建一个内存数据库，这只会在 session 的有效时间内持续。 如果文件名 filename 为实际的设备文件名称，那么 SQLite3::open() 将使用这个参数值尝试打开数据库文件。如果该名称的文件不存在，那么将创建一个新的命名为该名称的数据库文件。 可选的 flags 用于判断是否打开 SQLite 数据库。默认情况下，当使用 SQLITE3_OPEN_READWRITE | SQLITE3_OPEN_CREATE 时打开。
public bool SQLite3::exec (string $query)	该例程提供了一个执行 SQL 命令的快捷方式，SQL 命令由 sql 参数提供，可以由多个 SQL 命令组成。该程序用于对给定的数据库执行一个无结果的查询。
public SQLite3Result SQLite3::query (string $query)	该例程执行一个 SQL 查询，如果查询到返回结果则返回一个 **SQLite3Result** 对象。
public int SQLite3::lastErrorCode (void)	该例程返回最近一次失败的 SQLite 请求的数值结果代码。
public string SQLite3::lastErrorMsg (void)	该例程返回最近一次失败的 SQLite 请求的英语文本描述。
public int SQLite3::changes (void)	该例程返回最近一次的 SQL 语句更新或插入或删除的数据库行数。
public bool SQLite3::close (void)	该例程关闭之前调用 SQLite3::open() 打开的数据库连接。
public string SQLite3::escapeString (string $value)	该例程返回一个字符串，在 SQL 语句中，出于安全考虑，该字符串已被正确地转义。

连接数据库

下面的 PHP 代码显示了如何连接到一个现有的数据库。如果数据库不存在，那么它就会被创建，最后将返回一个数据库对象。

```
<?php
   class MyDB extends SQLite3
   {
      function __construct()
      {
         $this->open('test.db');
      }
   }
   $db = new MyDB();
   if(!$db){
      echo $db->lastErrorMsg();
   } else {
      echo "Opened database successfully\n";
   }
?>
```

现在，让我们来运行上面的程序，在当前目录中创建我们的数据库 **test.db**。您可以根据需要改变路径。如果数据库成功创建，那么会显示下面所示的消息：

```
Open database successfully
```

创建表

下面的 PHP 代码段将用于在先前创建的数据库中创建一个表：

```
<?php
   class MyDB extends SQLite3
   {
      function __construct()
      {
         $this->open('test.db');
      }
   }
   $db = new MyDB();
   if(!$db){
      echo $db->lastErrorMsg();
   } else {
      echo "Opened database successfully\n";
   }

   $sql =<<<EOF
      CREATE TABLE COMPANY
      (ID INT PRIMARY KEY     NOT NULL,
      NAME           TEXT    NOT NULL,
      AGE            INT     NOT NULL,
      ADDRESS        CHAR(50),
      SALARY         REAL);
EOF;

   $ret = $db->exec($sql);
   if(!$ret){
      echo $db->lastErrorMsg();
   } else {
      echo "Table created successfully\n";
   }
   $db->close();
?>
```

上述程序执行时，它会在 **test.db** 中创建 COMPANY 表，并显示下面所示的消息：

```
Opened database successfully
Table created successfully
```

INSERT 操作

下面的 PHP 程序显示了如何在上面创建的 COMPANY 表中创建记录：

```
<?php
   class MyDB extends SQLite3
   {
      function __construct()
      {
         $this->open('test.db');
      }
   }
   $db = new MyDB();
   if(!$db){
      echo $db->lastErrorMsg();
   } else {
      echo "Opened database successfully\n";
   }

   $sql =<<<EOF
      INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
      VALUES (1, 'Paul', 32, 'California', 20000.00 );

      INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
      VALUES (2, 'Allen', 25, 'Texas', 15000.00 );

      INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
      VALUES (3, 'Teddy', 23, 'Norway', 20000.00 );

      INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
      VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 );
EOF;

   $ret = $db->exec($sql);
   if(!$ret){
      echo $db->lastErrorMsg();
   } else {
      echo "Records created successfully\n";
   }
   $db->close();
?>
```

上述程序执行时，它会在 COMPANY 表中创建给定记录，并会显示以下两行：

```
Opened database successfully
Records created successfully
```

SELECT 操作

下面的 PHP 程序显示了如何从前面创建的 COMPANY 表中获取并显示记录：

```
<?php
   class MyDB extends SQLite3
   {
      function __construct()
      {
         $this->open('test.db');
      }
   }
   $db = new MyDB();
   if(!$db){
      echo $db->lastErrorMsg();
   } else {
      echo "Opened database successfully\n";
   }

   $sql =<<<EOF
      SELECT * from COMPANY;
EOF;

   $ret = $db->query($sql);
   while($row = $ret->fetchArray(SQLITE3_ASSOC) ){
      echo "ID = ". $row['ID'] . "\n";
      echo "NAME = ". $row['NAME'] ."\n";
      echo "ADDRESS = ". $row['ADDRESS'] ."\n";
      echo "SALARY =  ".$row['SALARY'] ."\n\n";
   }
   echo "Operation done successfully\n";
   $db->close();
?>
```

上述程序执行时，它会产生以下结果：

```
Opened database successfully
ID = 1
NAME = Paul
ADDRESS = California
SALARY =  20000

ID = 2
NAME = Allen
ADDRESS = Texas
SALARY =  15000

ID = 3
NAME = Teddy
ADDRESS = Norway
SALARY =  20000

ID = 4
NAME = Mark
ADDRESS = Rich-Mond
SALARY =  65000

Operation done successfully
```

UPDATE 操作

下面的 PHP 代码显示了如何使用 UPDATE 语句来更新任何记录，然后从 COMPANY 表中获取并显示更新的记录：

```
<?php
   class MyDB extends SQLite3
   {
      function __construct()
      {
         $this->open('test.db');
      }
   }
   $db = new MyDB();
   if(!$db){
      echo $db->lastErrorMsg();
   } else {
      echo "Opened database successfully\n";
   }
   $sql =<<<EOF
      UPDATE COMPANY set SALARY = 25000.00 where ID=1;
EOF;
   $ret = $db->exec($sql);
   if(!$ret){
      echo $db->lastErrorMsg();
   } else {
      echo $db->changes(), " Record updated successfully\n";
   }

   $sql =<<<EOF
      SELECT * from COMPANY;
EOF;
   $ret = $db->query($sql);
   while($row = $ret->fetchArray(SQLITE3_ASSOC) ){
      echo "ID = ". $row['ID'] . "\n";
      echo "NAME = ". $row['NAME'] ."\n";
      echo "ADDRESS = ". $row['ADDRESS'] ."\n";
      echo "SALARY =  ".$row['SALARY'] ."\n\n";
   }
   echo "Operation done successfully\n";
   $db->close();
?>
```

上述程序执行时，它会产生以下结果：

```
Opened database successfully
1 Record updated successfully
ID = 1
NAME = Paul
ADDRESS = California
SALARY =  25000

ID = 2
NAME = Allen
ADDRESS = Texas
SALARY =  15000

ID = 3
NAME = Teddy
ADDRESS = Norway
SALARY =  20000

ID = 4
NAME = Mark
ADDRESS = Rich-Mond
SALARY =  65000

Operation done successfully
```

DELETE 操作

下面的 PHP 代码显示了如何使用 DELETE 语句删除任何记录，然后从 COMPANY 表中获取并显示剩余的记录：

```
<?php
   class MyDB extends SQLite3
   {
      function __construct()
      {
         $this->open('test.db');
      }
   }
   $db = new MyDB();
   if(!$db){
      echo $db->lastErrorMsg();
   } else {
      echo "Opened database successfully\n";
   }
   $sql =<<<EOF
      DELETE from COMPANY where ID=2;
EOF;
   $ret = $db->exec($sql);
   if(!$ret){
     echo $db->lastErrorMsg();
   } else {
      echo $db->changes(), " Record deleted successfully\n";
   }

   $sql =<<<EOF
      SELECT * from COMPANY;
EOF;
   $ret = $db->query($sql);
   while($row = $ret->fetchArray(SQLITE3_ASSOC) ){
      echo "ID = ". $row['ID'] . "\n";
      echo "NAME = ". $row['NAME'] ."\n";
      echo "ADDRESS = ". $row['ADDRESS'] ."\n";
      echo "SALARY =  ".$row['SALARY'] ."\n\n";
   }
   echo "Operation done successfully\n";
   $db->close();
?>
```

上述程序执行时，它会产生以下结果：

```
Opened database successfully
1 Record deleted successfully
ID = 1
NAME = Paul
ADDRESS = California
SALARY =  25000

ID = 3
NAME = Teddy
ADDRESS = Norway
SALARY =  20000

ID = 4
NAME = Mark
ADDRESS = Rich-Mond
SALARY =  65000

Operation done successfully
```

SQLite - Perl

安装

SQLite3 可使用 Perl DBI 模块与 Perl 进行集成。Perl DBI 模块是 Perl 编程语言的数据库访问模块。它定义了一组提供标准数据库接口的方法、变量及规则。

下面显示了在 Linux/UNIX 机器上安装 DBI 模块的简单步骤：

```
$ wget http://search.cpan.org/CPAN/authors/id/T/TI/TIMB/DBI-1.625.t
$ tar xvfz DBI-1.625.tar.gz
$ cd DBI-1.625
$ perl Makefile.PL
$ make
$ make install
```

如果您需要为 DBI 安装 SQLite 驱动程序，那么可按照以下步骤进行安装：

```
$ wget http://search.cpan.org/CPAN/authors/id/M/MS/MSERGEANT/DBD-SQ
$ tar xvfz DBD-SQLite-1.11.tar.gz
$ cd DBD-SQLite-1.11
$ perl Makefile.PL
$ make
$ make install
```

DBI 接口 API

以下是重要的 DBI 程序，可以满足您在 Perl 程序中使用 SQLite 数据库的需求。如果您需要了解更多细节，请查看 Perl DBI 官方文档。

API	描述
DBI->connect($data_source, "", "", \%attr)	建立一个到被请求的 $data*source* 的数据库连接或者 *session*。如果连接成功，则返回一个数据库处理对象。 数据源形式如下所示：***DBI:SQLite:dbname='test.db'***。其中，*SQLite* 是 *SQLite* 驱动程序名称，*test.db* 是 *SQLite* 数据库文件的名称。如果文件名 *_filename* 赋值为 **':memory:'**，那么它将会在 RAM 中创建一个内存数据库，这只会在 session 的有效时间内持续。 如果文件名 filename 为实际的设备文件名称，那么它将使用这个参数值尝试打开数据库文件。如果该名称的文件不存在，那么将创建一个新的命名为该名称的数据库文件。您可以保留第二个和第三个参数为空白字符串，最后一个参数用于传递各种属性，详见下面的实例讲解。
$dbh->do($sql)	该例程准备并执行一个简单的 SQL 语句。返回受影响的行数，如果发生错误则返回 undef。返回值 -1 意味着行数未知，或不适用 ，或不可用。在这里，$dbh 是由 DBI->connect() 调用返回的处理。
$dbh->prepare($sql)	该例程为数据库引擎后续执行准备一个语句，并返回一个语句处理对象。
$sth->execute()	该例程执行任何执行预准备的语句需要的处理。如果发生错误则返回 undef。如果成功执行，则无论受影响的行数是多少，总是返回 true。在这里，$sth 是由 $dbh->prepare($sql) 调用返回的语句处理。
$sth->fetchrow_array()	该例程获取下一行数据，并以包含各字段值的列表形式返回。在该列表中，Null 字段将作为 undef 值返回。
$DBI::err	这相当于 $h->err。其中，$h 是任何的处理类型，比如 $dbh、$sth 或 $drh。该程序返回最后调用的驱动程序（driver）方法的数据库引擎错误代码。
$DBI::errstr	这相当于 $h->errstr。其中，$h 是任何的处理类型，比如 $dbh、$sth 或 $drh。该程序返回最后调用的 DBI 方法的数据库引擎错误消息。
$dbh->disconnect()	该例程关闭之前调用 DBI->connect() 打开的数据库连接。

连接数据库

下面的 Perl 代码显示了如何连接到一个现有的数据库。如果数据库不存在，那么它就会被创建，最后将返回一个数据库对象。

```
#!/usr/bin/perl

use DBI;
use strict;

my $driver   = "SQLite";
my $database = "test.db";
my $dsn = "DBI:$driver:dbname=$database";
my $userid = "";
my $password = "";
my $dbh = DBI->connect($dsn, $userid, $password, { RaiseError => 1
                      or die $DBI::errstr;

print "Opened database successfully\n";
```

现在，让我们来运行上面的程序，在当前目录中创建我们的数据库 **test.db**。您可以根据需要改变路径。保存上面代码到 sqlite.pl 文件中，并按如下所示执行。如果数据库成功创建，那么会显示下面所示的消息：

```
$ chmod +x sqlite.pl
$ ./sqlite.pl
Open database successfully
```

创建表

下面的 Perl 代码段将用于在先前创建的数据库中创建一个表：

```
#!/usr/bin/perl

use DBI;
use strict;

my $driver   = "SQLite";
my $database = "test.db";
my $dsn = "DBI:$driver:dbname=$database";
my $userid = "";
my $password = "";
my $dbh = DBI->connect($dsn, $userid, $password, { RaiseError => 1
                      or die $DBI::errstr;
print "Opened database successfully\n";

my $stmt = qq(CREATE TABLE COMPANY
      (ID INT PRIMARY KEY     NOT NULL,
       NAME           TEXT    NOT NULL,
       AGE            INT     NOT NULL,
       ADDRESS        CHAR(50),
       SALARY         REAL););
my $rv = $dbh->do($stmt);
if($rv < 0){
   print $DBI::errstr;
} else {
   print "Table created successfully\n";
}
$dbh->disconnect();
```

上述程序执行时，它会在 **test.db** 中创建 COMPANY 表，并显示下面所示的消息：

```
Opened database successfully
Table created successfully
```

注意：如果您在任何操作中遇到了下面的错误： in case you see following error in any of the operation:

```
DBD::SQLite::st execute failed: not an error(21) at dbdimp.c line 3
```

在这种情况下，您已经在 DBD-SQLite 安装中打开了可用的 dbdimp.c 文件，找到 **sqlite3_prepare()** 函数，并把它的第三个参数 0 改为 -1。最后使用 **make** 和 **make install** 安装 DBD::SQLite，即可解决问题。 in this case you will have open dbdimp.c file available in DBD-SQLite installation and find out **sqlite3_prepare()** function and change its third argument to -1 instead of 0. Finally install DBD::SQLite using **make** and do **make install** to resolve the problem.

INSERT 操作

下面的 Perl 程序显示了如何在上面创建的 COMPANY 表中创建记录：

```
#!/usr/bin/perl

use DBI;
use strict;

my $driver   = "SQLite";
my $database = "test.db";
my $dsn = "DBI:$driver:dbname=$database";
my $userid = "";
my $password = "";
my $dbh = DBI->connect($dsn, $userid, $password, { RaiseError => 1
                      or die $DBI::errstr;
print "Opened database successfully\n";

my $stmt = qq(INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
      VALUES (1, 'Paul', 32, 'California', 20000.00 ));
my $rv = $dbh->do($stmt) or die $DBI::errstr;

$stmt = qq(INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
      VALUES (2, 'Allen', 25, 'Texas', 15000.00 ));
$rv = $dbh->do($stmt) or die $DBI::errstr;

$stmt = qq(INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
      VALUES (3, 'Teddy', 23, 'Norway', 20000.00 ));
$rv = $dbh->do($stmt) or die $DBI::errstr;

$stmt = qq(INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
      VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 ););
$rv = $dbh->do($stmt) or die $DBI::errstr;

print "Records created successfully\n";
$dbh->disconnect();
```

上述程序执行时，它会在 COMPANY 表中创建给定记录，并会显示以下两行：

```
Opened database successfully
Records created successfully
```

SELECT 操作

下面的 Perl 程序显示了如何从前面创建的 COMPANY 表中获取并显示记录：

```
#!/usr/bin/perl

use DBI;
use strict;

my $driver   = "SQLite";
my $database = "test.db";
my $dsn = "DBI:$driver:dbname=$database";
my $userid = "";
my $password = "";
my $dbh = DBI->connect($dsn, $userid, $password, { RaiseError => 1
                      or die $DBI::errstr;
print "Opened database successfully\n";

my $stmt = qq(SELECT id, name, address, salary  from COMPANY;);
my $sth = $dbh->prepare( $stmt );
my $rv = $sth->execute() or die $DBI::errstr;
if($rv < 0){
   print $DBI::errstr;
}
while(my @row = $sth->fetchrow_array()) {
      print "ID = ". $row[0] . "\n";
      print "NAME = ". $row[1] ."\n";
      print "ADDRESS = ". $row[2] ."\n";
      print "SALARY =  ". $row[3] ."\n\n";
}
print "Operation done successfully\n";
$dbh->disconnect();
```

上述程序执行时，它会产生以下结果：

```
Opened database successfully
ID = 1
NAME = Paul
ADDRESS = California
SALARY =  20000

ID = 2
NAME = Allen
ADDRESS = Texas
SALARY =  15000

ID = 3
NAME = Teddy
ADDRESS = Norway
SALARY =  20000

ID = 4
NAME = Mark
ADDRESS = Rich-Mond
SALARY =  65000

Operation done successfully
```

UPDATE 操作

下面的 Perl 代码显示了如何使用 UPDATE 语句来更新任何记录，然后从 COMPANY 表中获取并显示更新的记录：

```
#!/usr/bin/perl

use DBI;
use strict;

my $driver   = "SQLite";
my $database = "test.db";
my $dsn = "DBI:$driver:dbname=$database";
my $userid = "";
my $password = "";
my $dbh = DBI->connect($dsn, $userid, $password, { RaiseError => 1
                      or die $DBI::errstr;
print "Opened database successfully\n";

my $stmt = qq(UPDATE COMPANY set SALARY = 25000.00 where ID=1;);
my $rv = $dbh->do($stmt) or die $DBI::errstr;
if( $rv < 0 ){
   print $DBI::errstr;
}else{
   print "Total number of rows updated : $rv\n";
}
$stmt = qq(SELECT id, name, address, salary  from COMPANY;);
my $sth = $dbh->prepare( $stmt );
$rv = $sth->execute() or die $DBI::errstr;
if($rv < 0){
   print $DBI::errstr;
}
while(my @row = $sth->fetchrow_array()) {
      print "ID = ". $row[0] . "\n";
      print "NAME = ". $row[1] ."\n";
      print "ADDRESS = ". $row[2] ."\n";
      print "SALARY =  ". $row[3] ."\n\n";
}
print "Operation done successfully\n";
$dbh->disconnect();
```

上述程序执行时，它会产生以下结果：

```
Opened database successfully
Total number of rows updated : 1
ID = 1
NAME = Paul
ADDRESS = California
SALARY =  25000

ID = 2
NAME = Allen
ADDRESS = Texas
SALARY =  15000

ID = 3
NAME = Teddy
ADDRESS = Norway
SALARY =  20000

ID = 4
NAME = Mark
ADDRESS = Rich-Mond
SALARY =  65000

Operation done successfully
```

DELETE 操作

下面的 Perl 代码显示了如何使用 DELETE 语句删除任何记录，然后从 COMPANY 表中获取并显示剩余的记录：

```
#!/usr/bin/perl

use DBI;
use strict;

my $driver   = "SQLite";
my $database = "test.db";
my $dsn = "DBI:$driver:dbname=$database";
my $userid = "";
my $password = "";
my $dbh = DBI->connect($dsn, $userid, $password, { RaiseError => 1
                      or die $DBI::errstr;
print "Opened database successfully\n";

my $stmt = qq(DELETE from COMPANY where ID=2;);
my $rv = $dbh->do($stmt) or die $DBI::errstr;
if( $rv < 0 ){
   print $DBI::errstr;
}else{
   print "Total number of rows deleted : $rv\n";
}
$stmt = qq(SELECT id, name, address, salary  from COMPANY;);
my $sth = $dbh->prepare( $stmt );
$rv = $sth->execute() or die $DBI::errstr;
if($rv < 0){
   print $DBI::errstr;
}
while(my @row = $sth->fetchrow_array()) {
      print "ID = ". $row[0] . "\n";
      print "NAME = ". $row[1] ."\n";
      print "ADDRESS = ". $row[2] ."\n";
      print "SALARY =  ". $row[3] ."\n\n";
}
print "Operation done successfully\n";
$dbh->disconnect();
```

上述程序执行时，它会产生以下结果：

```
Opened database successfully
Total number of rows deleted : 1
ID = 1
NAME = Paul
ADDRESS = California
SALARY =  25000

ID = 3
NAME = Teddy
ADDRESS = Norway
SALARY =  20000

ID = 4
NAME = Mark
ADDRESS = Rich-Mond
SALARY =  65000

Operation done successfully
```

SQLite - Python

安装

SQLite3 可使用 sqlite3 模块与 Python 进行集成。sqlite3 模块是由 Gerhard Haring 编写的。它提供了一个与 PEP 249 描述的 DB-API 2.0 规范兼容的 SQL 接口。您不需要单独安装该模块，因为 Python 2.5.x 以上版本默认自带了该模块。

为了使用 sqlite3 模块，您首先必须创建一个表示数据库的连接对象，然后您可以有选择地创建光标对象，这将帮助您执行所有的 SQL 语句。

Python sqlite3 模块 API

以下是重要的 sqlite3 模块程序，可以满足您在 Python 程序中使用 SQLite 数据库的需求。如果您需要了解更多细节，请查看 Python sqlite3 模块的官方文档。

API	描述
sqlite3.connect(database [,timeout ,other optional arguments])	该 API 打开一个到 SQLite 数据库文件 database 的链接。您可以使用 ":memory:" 来在 RAM 中打开一个到 database 的数据库连接，而不是在磁盘上打开。如果数据库成功打开，则返回一个连接对象。 当一个数据库被多个连接访问，且其中一个修改了数据库，此时 SQLite 数据库被锁定，直到事务提交。timeout 参数表示连接等待锁定的持续时间，直到发生异常断开连接。timeout 参数默认是 5.0（5 秒）。 如果给定的数据库名称 filename 不存在，则该调用将创建一个数据库。如果您不想在当前目录中创建数据库，那么您可以指定带有路径的文件名，这样您就能在任意地方创建数据库。
connection.cursor([cursorClass])	该例程创建一个 **cursor**，将在 Python 数据库编程中用到。该方法接受一个单一的可选的参数 cursorClass。如果提供了

	该参数，则它必须是一个扩展自 sqlite3.Cursor 的自定义的 cursor 类。
cursor.execute(sql [, optional parameters])	该例程执行一个 SQL 语句。该 SQL 语句可以被参数化（即使用占位符代替 SQL 文本）。sqlite3 模块支持两种类型的占位符：问号和命名占位符（命名样式）。 例如：cursor.execute("insert into people values (?, ?)", (who, age))
connection.execute(sql [, optional parameters])	该例程是上面执行的由光标（cursor）对象提供的方法的快捷方式，它通过调用光标（cursor）方法创建了一个中间的光标对象，然后通过给定的参数调用光标的 execute 方法。
cursor.executemany(sql, seq_of_parameters)	该例程对 seq_of_parameters 中的所有参数或映射执行一个 SQL 命令。
connection.executemany(sql[, parameters])	该例程是一个由调用光标（cursor）方法创建的中间的光标对象的快捷方式，然后通过给定的参数调用光标的 executemany 方法。
cursor.executescript(sql_script)	该例程一旦接收到脚本，会执行多个 SQL 语句。它首先执行 COMMIT 语句，然后执行作为参数传入的 SQL 脚本。所有的 SQL 语句应该用分号（;）分隔。
connection.executescript(sql_script)	该例程是一个由调用光标（cursor）方法创建的中间的光标对象的快捷方式，然后通过给定的参数调用光标的 executescript 方法。
connection.total_changes()	该例程返回自数据库连接打开以来被修改、插入或删除的数据库总行数。
	该方法提交当前的食物。如果您未调用该方法，那么自您上

connection.commit()	一次调用 commit() 以来所做的任何动作对其他数据库连接来说是不可见的。
connection.rollback()	该方法回滚自上一次调用 commit() 以来对数据库所做的更改。
connection.close()	该方法关闭数据库连接。请注意，这不会自动调用 commit()。如果您之前未调用 commit() 方法，就直接关闭数据库连接，您所做的所有更改将全部丢失！
cursor.fetchone()	该方法获取查询结果集中的下一行，返回一个单一的序列，当没有更多可用的数据时，则返回 None。
cursor.fetchmany([size=cursor.arraysize])	该方法获取查询结果集中的下一行组，返回一个列表。当没有更多的可用的行时，则返回一个空的列表。该方法尝试获取由 size 参数指定的尽可能多的行。
cursor.fetchall()	该例程获取查询结果集中所有（剩余）的行，返回一个列表。当没有可用的行时，则返回一个空的列表。

连接数据库

下面的 Python 代码显示了如何连接到一个现有的数据库。如果数据库不存在，那么它就会被创建，最后将返回一个数据库对象。

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')

print "Opened database successfully";
```

在这里，您也可以把数据库名称复制为特定的名称 **:memory:**，这样就会在 RAM 中创建一个数据库。现在，让我们来运行上面的程序，在当前目录中创建我们的数据库 **test.db**。您可以根据需要改变路径。保存上面代码到 sqlite.py 文件中，并按

如下所示执行。如果数据库成功创建，那么会显示下面所示的消息：

```
$chmod +x sqlite.py
$./sqlite.py
Open database successfully
```

创建表

下面的 Python 代码段将用于在先前创建的数据库中创建一个表：

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')
print "Opened database successfully";

conn.execute('''CREATE TABLE COMPANY
       (ID INT PRIMARY KEY     NOT NULL,
       NAME           TEXT    NOT NULL,
       AGE            INT     NOT NULL,
       ADDRESS        CHAR(50),
       SALARY         REAL);''')
print "Table created successfully";

conn.close()
```

上述程序执行时，它会在 **test.db** 中创建 COMPANY 表，并显示下面所示的消息：

```
Opened database successfully
Table created successfully
```

INSERT 操作

下面的 Python 程序显示了如何在上面创建的 COMPANY 表中创建记录：

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')
print "Opened database successfully";

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (1, 'Paul', 32, 'California', 20000.00 )");

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (2, 'Allen', 25, 'Texas', 15000.00 )");

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (3, 'Teddy', 23, 'Norway', 20000.00 )");

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 )");

conn.commit()
print "Records created successfully";
conn.close()
```

上述程序执行时，它会在 COMPANY 表中创建给定记录，并会显示以下两行：

```
Opened database successfully
Records created successfully
```

SELECT 操作

下面的 Python 程序显示了如何从前面创建的 COMPANY 表中获取并显示记录：

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')
print "Opened database successfully";

cursor = conn.execute("SELECT id, name, address, salary  from COMPA
for row in cursor:
   print "ID = ", row[0]
   print "NAME = ", row[1]
   print "ADDRESS = ", row[2]
   print "SALARY = ", row[3], "\n"

print "Operation done successfully";
conn.close()
```

上述程序执行时，它会产生以下结果：

```
Opened database successfully
ID =  1
NAME =  Paul
ADDRESS =  California
SALARY =  20000.0

ID =  2
NAME =  Allen
ADDRESS =  Texas
SALARY =  15000.0

ID =  3
NAME =  Teddy
ADDRESS =  Norway
SALARY =  20000.0

ID =  4
NAME =  Mark
ADDRESS =  Rich-Mond
SALARY =  65000.0

Operation done successfully
```

UPDATE 操作

下面的 Python 代码显示了如何使用 UPDATE 语句来更新任何记录，然后从 COMPANY 表中获取并显示更新的记录：

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')
print "Opened database successfully";

conn.execute("UPDATE COMPANY set SALARY = 25000.00 where ID=1")
conn.commit
print "Total number of rows updated :", conn.total_changes

cursor = conn.execute("SELECT id, name, address, salary  from COMPA
for row in cursor:
   print "ID = ", row[0]
   print "NAME = ", row[1]
   print "ADDRESS = ", row[2]
   print "SALARY = ", row[3], "\n"

print "Operation done successfully";
conn.close()
```

上述程序执行时，它会产生以下结果：

```
Opened database successfully
Total number of rows updated : 1
ID =  1
NAME =  Paul
ADDRESS =  California
SALARY =  25000.0

ID =  2
NAME =  Allen
ADDRESS =  Texas
SALARY =  15000.0

ID =  3
NAME =  Teddy
ADDRESS =  Norway
SALARY =  20000.0

ID =  4
NAME =  Mark
ADDRESS =  Rich-Mond
SALARY =  65000.0

Operation done successfully
```

DELETE 操作

下面的 Python 代码显示了如何使用 DELETE 语句删除任何记录，然后从 COMPANY 表中获取并显示剩余的记录：

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')
print "Opened database successfully";

conn.execute("DELETE from COMPANY where ID=2;")
conn.commit
print "Total number of rows deleted :", conn.total_changes

cursor = conn.execute("SELECT id, name, address, salary  from COMPA
for row in cursor:
   print "ID = ", row[0]
   print "NAME = ", row[1]
   print "ADDRESS = ", row[2]
   print "SALARY = ", row[3], "\n"

print "Operation done successfully";
conn.close()
```

上述程序执行时，它会产生以下结果：

```
Opened database successfully
Total number of rows deleted : 1
ID =  1
NAME =  Paul
ADDRESS =  California
SALARY =  20000.0

ID =  3
NAME =  Teddy
ADDRESS =  Norway
SALARY =  20000.0

ID =  4
NAME =  Mark
ADDRESS =  Rich-Mond
SALARY =  65000.0

Operation done successfully
```

MySQL 教程

MySQL 教程



Mysql是最流行的关系型数据库管理系统，在WEB应用方面MySQL是最好的RDBMS(Relational Database Management System：关系数据库管理系统)应用软件之一。

在本教程中，会让大家快速掌握Mysql的基本知识，并轻松使用Mysql数据库。

什么是数据库？

数据库（Database）是按照数据结构来组织、存储和管理数据的仓库，

每个数据库都有一个或多个不同的API用于创建，访问，管理，搜索和复制所保存的数据。

我们也可以将数据存储在文件中，但是在文件中读写数据速度相对较慢。

所以，现在我们使用关系型数据库管理系统（RDBMS）来存储和管理的大数据量。所谓的关系型数据库，是建立在关系模型基础上的数据库，借助于集合代数等数学概念和方法来处理数据库中的数据。

RDBMS即关系数据库管理系统(Relational Database Management System)的特点：

- 1.数据以表格的形式出现
- 2.每行为各种记录名称
- 3.每列为记录名称所对应的数据域
- 4.许多的行和列组成一张表单
- 5.若干的表单组成database

RDBMS 术语

在我们开始学习MySQL 数据库前，让我们先了解下RDBMS的一些术语：

- 数据库: 数据库是一些关联表的集合。.

- 数据表: 表是数据的矩阵。在一个数据库中的表看起来像一个简单的电子表格。
- 列: 一列(数据元素) 包含了相同的数据, 例如邮政编码的数据。
- 行：一行（=元组，或记录）是一组相关的数据，例如一条用户订阅的数据。
- 冗余：存储两倍数据，冗余可以使系统速度更快。
- 主键：主键是唯一的。一个数据表中只能包含一个主键。你可以使用主键来查询数据。
- 外键：外键用于关联两个表。
- 复合键：复合键（组合键）将多个列作为一个索引键，一般用于复合索引。
- 索引：使用索引可快速访问数据库表中的特定信息。索引是对数据库表中一列或多列的值进行排序的一种结构。类似于书籍的目录。
- 参照完整性: 参照的完整性要求关系中不允许引用不存在的实体。与实体完整性是关系模型必须满足的完整性约束条件，目的是保证数据的一致性。

Mysql数据库

MySQL是一个关系型数据库管理系统，由瑞典MySQL AB公司开发，目前属于Oracle公司。MySQL是一种关联数据库管理系统，关联数据库将数据保存在不同的表中，而不是将所有数据放在一个大仓库内，这样就增加了速度并提高了灵活性。

- Mysql是开源的，所以你不需要支付额外的费用。
- Mysql支持大型的数据库。可以处理拥有上千万条记录的大型数据库。
- MySQL使用标准的SQL数据语言形式。
- Mysql可以允许于多个系统上，并且支持多种语言。这些编程语言包括C、C++、Python、Java、Perl、PHP、Eiffel、Ruby和Tcl等。
- Mysql对PHP有很好的支持，PHP是目前最流行的Web开发语言。
- MySQL支持大型数据库，支持5000万条记录的数据仓库，32位系统表文件最大可支持4GB，64位系统支持最大的表文件为8TB。
- Mysql是可以定制的，采用了GPL协议，你可以修改源码来开发自己的Mysql系统。

在开始学习本教程前你应该了解？

在开始学习本教程前你应该了解PHP和HTML的基础知识，并能简单的应用。

本教程的很多例子都跟PHP语言有关，我们的实例基本上是采用PHP语言来演示。

如果你还不了解PHP，你可以通过本站的PHP教程来了解该语言。

MySQL 安装

所有平台的Mysql下载地址为： MySQL 下载. 挑选你需要的 *MySQL Community Server* 版本及对应的平台。

Linux/UNIX上安装Mysql

Linux平台上推荐使用RPM包来安装Mysql,MySQL AB提供了以下RPM包的下载地址：

- **MySQL** - MySQL服务器。你需要该选项，除非你只想连接运行在另一台机器上的MySQL服务器。
- **MySQL-client** - MySQL 客户端程序，用于连接并操作Mysql服务器。
- **MySQL-devel** - 库和包含文件，如果你想要编译其它MySQL客户端，例如Perl模块，则需要安装该RPM包。
- **MySQL-shared** - 该软件包包含某些语言和应用程序需要动态装载的共享库(libmysqlclient.so*)，使用MySQL。
- **MySQL-bench** - MySQL数据库服务器的基准和性能测试工具。

以下安装Mysql RMP的实例是在SuSE Linux系统上进行，当然该安装步骤也适合应用于其他支持RPM的Linux系统，如:Centos。

安装步骤如下：

使用root用户登陆你的Linux系统。

下载Mysql RPM包，下载地址为：MySQL 下载。

通过以下命令执行Mysql安装，rpm包为你下载的rpm包：

```
[root@host]# rpm -i MySQL-5.0.9-0.i386.rpm
```

以上安装mysql服务器的过程会创建mysql用户，并创建一个mysql配置文件my.cnf。

你可以在/usr/bin和/usr/sbin中找到所有与MySQL相关的二进制文件。所有数据表和数据库将在/var/lib/mysql目录中创建。

以下是一些mysql可选包的安装过程，你可以根据自己的需要来安装：

```
[root@host]# rpm -i MySQL-client-5.0.9-0.i386.rpm
[root@host]# rpm -i MySQL-devel-5.0.9-0.i386.rpm
[root@host]# rpm -i MySQL-shared-5.0.9-0.i386.rpm
[root@host]# rpm -i MySQL-bench-5.0.9-0.i386.rpm
```

Window上安装Mysql

Window上安装Mysql相对来说会较为简单，你只需要载 MySQL 下载中下载window版本的mysql安装包，并解压安装包。

双击 setup.exe 文件，接下来你只需要安装默认的配置点击"next"即可，默认情况下安装信息会在C:\mysql目录中。

接下来你可以通过"开始" =》在搜索框中输入 " cmd" 命令 =》 在命令提示符上切换到 C:\mysql\bin 目录，并输入一下命令：

```
mysqld.exe --console
```

如果安装成功以上命令将输出一些mysql启动及InnoDB信息。

验证Mysql安装

在成功安装Mysql后，一些基础表会表初始化，在服务器启动后，你可以通过简单的测试来验证Mysql是否工作正常。

使用 mysqladmin 工具来获取服务器状态：

使用 mysqladmin 命令俩检查服务器的版本,在linux上该二进制文件位于 /usr/bin on linux ，在window上该二进制文件位于C:\mysql\bin 。

```
[root@host]# mysqladmin --version
```

linux上该命令将输出以下结果，该结果基于你的系统信息：

```
mysqladmin  Ver 8.23 Distrib 5.0.9-0, for redhat-linux-gnu on i386
```

如果以上命令执行后未输入任何信息，说明你的Mysql未安装成功。

使用 MySQL Client(Mysql客户端) 执行简单的SQL命令

你可以在 MySQL Client(Mysql客户端) 使用 mysql 命令连接到Mysql服务器上，默认情况下Mysql服务器的密码为空，所以本实例不需要输入密码。

命令如下：

```
[root@host]# mysql
```

以上命令执行后会输出 mysql>提示符，这说明你已经成功连接到Mysql服务器上，你可以在 mysql> 提示符执行SQL命令：

```
mysql> SHOW DATABASES;
+----------+
| Database |
+----------+
| mysql    |
| test     |
+----------+
2 rows in set (0.13 sec)
```

Mysql安装后需要做的

Mysql安装成功后，默认的root用户密码为空，你可以使用以下命令来创建root用户的密码：

```
[root@host]# mysqladmin -u root password "new_password";
```

现在你可以通过以下命令来连接到Mysql服务器：

```
[root@host]# mysql -u root -p
Enter password:*******
```

注意：在输入密码时，密码是不会显示了，你正确输入即可。

Linux系统启动时启动 MySQL

如果你需要在Linux系统启动时启动 MySQL 服务器，你需要在 /etc/rc.local 文件中添加以下命令：

```
/etc/init.d/mysqld start
```

同样，你需要将 mysqld 二进制文件添加到 /etc/init.d/ 目录中。

MySQL 管理

启动及关闭 MySQL 服务器

首先，我们需要通过以下命令来检查MySQL服务器是否启动：

```
ps -ef | grep mysqld
```

如果MySql已经启动，以上命令将输出mysql进程列表， 如果mysql未启动，你可以使用以下命令来启动mysql服务器:

```
root@host# cd /usr/bin
./safe_mysqld &
```

如果你想关闭目前运行的 MySQL 服务器, 你可以执行以下命令:

```
root@host# cd /usr/bin
./mysqladmin -u root -p shutdown
Enter password: ******
```

MySQL 用户设置

如果你需要添加 MySQL 用户，你只需要在 mysql 数据库中的 user 表添加新用户即可。

以下为添加用户的的实例，用户名为guest，密码为guest123，并授权用户可进行 SELECT, INSERT 和 UPDATE操作权限：

```
root@host# mysql -u root -p
Enter password:*******
mysql> use mysql;
Database changed

mysql> INSERT INTO user
          (host, user, password,
           select_priv, insert_priv, update_priv)
           VALUES ('localhost', 'guest',
           PASSWORD('guest123'), 'Y', 'Y', 'Y');
Query OK, 1 row affected (0.20 sec)

mysql> FLUSH PRIVILEGES;
Query OK, 1 row affected (0.01 sec)

mysql> SELECT host, user, password FROM user WHERE user = 'guest';
+-----------+---------+------------------+
| host      | user    | password         |
+-----------+---------+------------------+
| localhost | guest | 6f8c114b58f2ce9e |
+-----------+---------+------------------+
1 row in set (0.00 sec)
```

在添加用户时，请注意使用MySQL提供的 PASSWORD() 函数来对密码进行加密。你可以在以上实例看到用户密码加密后为： 6f8c114b58f2ce9e.

注意：在注意需要执行 FLUSH PRIVILEGES 语句。 这个命令执行后会重新载入授权表。 如果你不使用该命令，你就无法使用新创建的用户来连接mysql服务器，除非你重启mysql服务器。

你可以在创建用户时，为用户指定权限，在对应的权限列中，在插入语句中设置为 'Y' 即可，用户权限列表如下：

- Select_priv
- Insert_priv
- Update_priv
- Delete_priv
- Create_priv
- Drop_priv
- Reload_priv
- Shutdown_priv
- Process_priv
- File_priv
- Grant_priv
- References_priv
- Index_priv
- Alter_priv

另外一种添加用户的方法为通过SQL的 GRANT 命令，你下命令会给指定数据库TUTORIALS添加用户 zara，密码为 zara123 。

```
root@host# mysql -u root -p password;
Enter password:*******
mysql> use mysql;
Database changed

mysql> GRANT SELECT,INSERT,UPDATE,DELETE,CREATE,DROP
    -> ON TUTORIALS.*
    -> TO 'zara'@'localhost'
    -> IDENTIFIED BY 'zara123';
```

以上命令会在mysql数据库中的user表创建一条用户信息记录。

注意: MySQL 的SQL语句以分号 (;) 作为结束标识。

/etc/my.cnf 文件配置

一般情况下，你不需要修改该配置文件，该文件默认配置如下：

```
[mysqld]
datadir=/var/lib/mysql
socket=/var/lib/mysql/mysql.sock

[mysql.server]
user=mysql
basedir=/var/lib

[safe_mysqld]
err-log=/var/log/mysqld.log
pid-file=/var/run/mysqld/mysqld.pid
```

在配置文件中，你可以指定不同的错误日志文件存放的目录，一般你不需要改动这些配置。

管理MySQL的命令

以下列出了使用Mysql数据库过程中常用的命令：

- **USE** 数据库名 :选择要操作的Mysql数据库，使用该命令后所有Mysql命令都只针对该数据库。
- **SHOW DATABASES:** 列出 MySQL 数据库管理系统的数据库列表。
- **SHOW TABLES:** 显示指定数据库的所有表，使用该命令前需要使用 use 命令来选择要操作的数据库。
- **SHOW COLUMNS FROM** 数据表**:** 显示数据表的属性，属性类型，主键信息

，是否为 NULL，默认值等其他信息。

- **SHOW INDEX FROM** 数据表**:** 显示数据表的详细索引信息，包括PRIMARY KEY（主键）。
- **SHOW TABLE STATUS LIKE** 数据表**\G:** 该命令将输出Mysql数据库管理系统的性能及统计信息。

MySQL PHP 语法

MySQL 可应用于多种语言，包括 PERL, C, C++, JAVA 和 PHP。 在这些语言中，Mysql在PHP的web开发中是应用最广泛。

在本教程中我们大部分实例都采用了PHP语言。你过你想了解Mysql在PHP中的应用，可以访问我们的 PHP中使用Mysql介绍。

PHP提供了多种方式来访问和操作Mysql数据库记录。PHP Mysql函数格式如下：

```
mysql_function(value,value,...);
```

以上格式中 function部分描述了mysql函数的功能，如

```
mysqli_connect($connect);
mysqli_query($connect,"SQL statement");
mysql_fetch_array()
mysql_connect(),mysql_close()
```

以下实例展示了PHP调用mysql函数的语法：

```
<html>
<head>
<title>PHP with MySQL</title>
</head>
<body>
<?php
   $retval = mysql_function(value, [value,...]);
   if( !$retval )
   {
       die ( "Error: a related error message" );
   }
   // Otherwise MySQL  or PHP Statements
?>
</body>
</html>
```

从下一章开始，我们将学习到更多的MySQL功能函数。

MySQL 连接

使用mysql二进制方式连接

您可以使用MySQL二进制方式进入到mysql命令提示符下来连接MySQL数据库。

实例

以下是从命令行中连接mysql服务器的简单实例：

```
[root@host]# mysql -u root -p
Enter password:******
```

在登录成功后会出现 mysql> 命令提示窗口，你可以在上面执行任何 SQL 语句。

以上命令执行后，登录成功输出结果如下:

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2854760 to server version: 5.0.9

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
```

在以上实例中，我们使用了root用户登录到mysql服务器，当然你也可以使用其他mysql用户登录。

如果用户权限足够，任何用户都可以在mysql的命令提示窗口中进行SQL操作。

退出 mysql> 命令提示窗口可以使用 exit 命令，如下所示：

```
mysql> exit
Bye
```

使用 PHP 脚本连接 MySQL

PHP 提供了 mysql_connect() 函数来连接数据库。

该函数有5个参数，在成功链接到MySQL后返回连接标识，失败返回 FALSE 。

语法

```
connection mysql_connect(server,user,passwd,new_link,client_flag);
```

参数说明：

参数	描述
server	可选。规定要连接的服务器。 可以包括端口号，例如 "hostname:port"，或者到本地套接字的路径，例如对于 localhost 的 ":/path/to/socket"。 如果 PHP 指令 mysql.default_host 未定义（默认情况），则默认值是 'localhost:3306'。
user	可选。用户名。默认值是服务器进程所有者的用户名。
passwd	可选。密码。默认值是空密码。
new_link	可选。如果用同样的参数第二次调用 mysql_connect()，将不会建立新连接，而将返回已经打开的连接标识。参数 new_link 改变此行为并使 mysql_connect() 总是打开新的连接，甚至当 mysql_connect() 曾在前面被用同样的参数调用过。
client_flag	可选。client_flags 参数可以是以下常量的组合： MYSQL_CLIENT_SSL - 使用 SSL 加密 MYSQL_CLIENT_COMPRESS - 使用压缩协议 MYSQL_CLIENT_IGNORE_SPACE - 允许函数名后的间隔 MYSQL_CLIENT_INTERACTIVE - 允许关闭连接之前的交互超时非活动时间

你可以使用PHP的 mysql_close() 函数来断开与MySQL数据库的链接。

该函数只有一个参数为mysql_connect()函数创建连接成功后返回的 MySQL 连接标识符。

语法

```
bool mysql_close ( resource $link_identifier );
```

本函数关闭指定的连接标识所关联的到 MySQL 服务器的非持久连接。如果没有指定 link_identifier，则关闭上一个打开的连接。

提示：通常不需要使用 mysql_close()，因为已打开的非持久连接会在脚本执行完毕后自动关闭。

注释：mysql_close() 不会关闭由 mysql_pconnect() 建立的持久连接。

实例

你可以尝试以下实例来连接到你的 MySQL 服务器:

```
<html>
<head>
<title>Connecting MySQL Server</title>
</head>
<body>
<?php
   $dbhost = 'localhost:3036';  //mysql服务器主机地址
   $dbuser = 'guest';      //mysql用户名
   $dbpass = 'guest123';//mysql用户名密码
   $conn = mysql_connect($dbhost, $dbuser, $dbpass);
   if(! $conn )
   {
     die('Could not connect: ' . mysql_error());
   }
   echo 'Connected successfully';
   mysql_close($conn);
?>
</body>
</html>
```

MySQL 创建数据库

使用 mysqladmin 创建数据库

使用普通用户，你可能需要特定的权限来创建或者删除 MySQL 数据库。

所以我们这边使用root用户登录，root用户拥有最高权限，可以使用 mysql mysqladmin 命令来创建数据库。

实例

以下命令简单的演示了创建数据库的过程，数据名为 TUTORIALS:

```
[root@host]# mysqladmin -u root -p create TUTORIALS
Enter password:******
```

以上命令执行成功后会创建 MySQL 数据库 TUTORIALS。

使用 PHP脚本 创建数据库

PHP使用 mysql_query 函数来创建或者删除 MySQL 数据库。

该函数有两个参数，在执行成功时返回 TRUE，否则返回 FALSE。

语法

```
bool mysql_query( sql, connection );
```

参数	描述
sql	必需。规定要发送的 SQL 查询。注释：查询字符串不应以分号结束。
connection	可选。规定 SQL 连接标识符。如果未规定，则使用上一个打开的连接。

实例

以下实例演示了使用PHP来创建一个数据库：

```
<html>
<head>
<title>Creating MySQL Database</title>
</head>
<body>
<?php
$dbhost = 'localhost:3036';
$dbuser = 'root';
$dbpass = 'rootpassword';
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if(! $conn )
{
  die('Could not connect: ' . mysql_error());
}
echo 'Connected successfully<br />';
$sql = 'CREATE DATABASE TUTORIALS';
$retval = mysql_query( $sql, $conn );
if(! $retval )
{
  die('Could not create database: ' . mysql_error());
}
echo "Database TUTORIALS created successfully\n";
mysql_close($conn);
?>
</body>
</html>
```

MySQL 删除数据库

使用 mysqladmin 删除数据库

使用普通用户登陆mysql服务器，你可能需要特定的权限来创建或者删除 MySQL 数据库。

所以我们这边使用root用户登录，root用户拥有最高权限，可以使用 mysql mysqladmin 命令来创建数据库。

在删除数据库过程中，务必要十分谨慎，因为在执行删除命令后，所有数据将会消失。

以下实例删除数据库TUTORIALS(该数据库在前一章节已创建)：

```
[root@host]# mysqladmin -u root -p drop TUTORIALS
Enter password:******
```

执行以上删除数据库命令后，会出现一个提示框，来确认是否真的删除数据库：

```
Dropping the database is potentially a very bad thing to do.
Any data stored in the database will be destroyed.

Do you really want to drop the 'TUTORIALS' database [y/N] y
Database "TUTORIALS" dropped
```

使用PHP脚本删除数据库

PHP使用 mysql_query 函数来创建或者删除 MySQL 数据库。

该函数有两个参数，在执行成功时返回 TRUE，否则返回 FALSE。

语法

```
bool mysql_query( sql, connection );
```

参数	描述
sql	必需。规定要发送的 SQL 查询。注释：查询字符串不应以分号结束。
connection	可选。规定 SQL 连接标识符。如果未规定，则使用上一个打开的连接。

实例

以下实例演示了使用PHP mysql_query函数来删除数据库：

```
<html>
<head>
<title>Deleting MySQL Database</title>
</head>
<body>
<?php
$dbhost = 'localhost:3036';
$dbuser = 'root';
$dbpass = 'rootpassword';
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if(! $conn )
{
  die('连接失败: ' . mysql_error());
}
echo '连接成功<br />';
$sql = 'DROP DATABASE TUTORIALS';
$retval = mysql_query( $sql, $conn );
if(! $retval )
{
  die('删除数据库失败: ' . mysql_error());
}
echo "数据库 TUTORIALS 删除成功\n";
mysql_close($conn);
?>
</body>
</html>
```

注意： 在使用PHP脚本删除数据库时，不会出现确认是否删除信息，会直接删除指定数据库，所以你在删除数据库时要特别小心。

MySQL 选择数据库

在你连接到 MySQL 数据库后，可能有多个可以操作的数据库，所以你需要选择你要操作的数据库。

从命令提示窗口中选择MySQL数据库

在 mysql> 提示窗口中可以很简单的选择特定的数据库。你可以使用SQL命令来选择指定的数据库。

实例

以下实例选取了数据库 TUTORIALS:

```
[root@host]# mysql -u root -p
Enter password:******
mysql> use TUTORIALS;
Database changed
mysql>
```

执行以上命令后，你就已经成功选择了 TUTORIALS 数据库，在后续的操作中都会在 TUTORIALS 数据库中执行。

注意:所有的数据库名，表名，表字段都是区分大小写的。所以你在使用SQL命令时需要输入正确的名称。

使用PHP脚本选择MySQL数据库

PHP 提供了函数 mysql_select_db 来选取一个数据库。函数在执行成功后返回 TRUE ，否则返回 FALSE 。

语法

```
bool mysql_select_db( db_name, connection );
```

参数	描述
db_name	必需。规定要选择的数据库。
connection	可选。规定 MySQL 连接。如果未指定，则使用上一个连接。

实例

以下实例展示了如何使用 mysql_select_db 函数来选取一个数据库：

```
<html>
<head>
<title>Selecting MySQL Database</title>
</head>
<body>
<?php
$dbhost = 'localhost:3036';
$dbuser = 'guest';
$dbpass = 'guest123';
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if(! $conn )
{
  die('Could not connect: ' . mysql_error());
}
echo 'Connected successfully';
mysql_select_db( 'TUTORIALS' );
mysql_close($conn);
?>
</body>
</html>
```

MySQL 数据类型

MySQL中定义数据字段的类型对你数据库的优化是非常重要的。

MySQL支持多种类型，大致可以分为三类：数值、日期/时间和字符串(字符)类型。

数值类型

MySQL支持所有标准SQL数值数据类型。

这些类型包括严格数值数据类型(INTEGER、SMALLINT、DECIMAL和NUMERIC)，以及近似数值数据类型(FLOAT、REAL和DOUBLE PRECISION)。

关键字INT是INTEGER的同义词，关键字DEC是DECIMAL的同义词。

BIT数据类型保存位字段值，并且支持MyISAM、MEMORY、InnoDB和BDB表。

作为SQL标准的扩展，MySQL也支持整数类型TINYINT、MEDIUMINT和BIGINT。下面的表显示了需要的每个整数类型的存储和范围。

类型	大小	范围（有符号）	范围（无符号）	用途
TINYINT	1 字节	(-128，127)	(0，255)	小整数值
SMALLINT	2 字节	(-32 768，32 767)	(0，65 535)	大整数值
MEDIUMINT	3 字节	(-8 388 608，8 388 607)	(0，16 777 215)	大整数值
INT或INTEGER	4 字节	(-2 147 483 648，2 147 483 647)	(0，4 294 967 295)	大整数值
BIGINT	8 字节	(-9 233 372 036 854 775 808，9 223 372 036 854 775 807)	(0，18 446 744 073 709 551 615)	极大整数值

FLOAT	4 字节	(-3.402 823 466 E+38，1.175 494 351 E-38)，0，(1.175 494 351 E-38，3.402 823 466 351 E+38)	0，(1.175 494 351 E-38，3.402 823 466 E+38)	单精度浮点数值
DOUBLE	8 字节	(1.797 693 134 862 315 7 E+308，2.225 073 858 507 201 4 E-308)，0，(2.225 073 858 507 201 4 E-308，1.797 693 134 862 315 7 E+308)	0，(2.225 073 858 507 201 4 E-308，1.797 693 134 862 315 7 E+308)	双精度浮点数值
DECIMAL	对DECIMAL(M,D)，如果M>D，为M+2否则为D+2	依赖于M和D的值	依赖于M和D的值	小数值

日期和时间类型

表示时间值的日期和时间类型为DATETIME、DATE、TIMESTAMP、TIME和YEAR。

每个时间类型有一个有效值范围和一个"零"值，当指定不合法的MySQL不能表示的值时使用"零"值。

TIMESTAMP类型有专有的自动更新特性，将在后面描述。

类型	大小(字节)	范围	格式	用途
DATE	3	1000-01-01/9999-12-31	YYYY-MM-DD	日期值
TIME	3	'-838:59:59'/'838:59:59'	HH:MM:SS	时间值或持续时间
YEAR	1	1901/2155	YYYY	年份值
DATETIME	8	1000-01-01 00:00:00/9999-12-31 23:59:59	YYYY-MM-DD HH:MM:SS	混合日期和时间值
TIMESTAMP	8	1970-01-01 00:00:00/2037 年某时	YYYYMMDD HHMMSS	混合日期和时间值，时间戳

字符串类型

字符串类型指CHAR、VARCHAR、BINARY、VARBINARY、BLOB、TEXT、ENUM和SET。该节描述了这些类型如何工作以及如何在查询中使用这些类型。

类型	大小	用途
CHAR	0-255字节	定长字符串
VARCHAR	0-255字节	变长字符串
TINYBLOB	0-255字节	不超过 255 个字符的二进制字符串
TINYTEXT	0-255字节	短文本字符串
BLOB	0-65 535字节	二进制形式的长文本数据
TEXT	0-65 535字节	长文本数据
MEDIUMBLOB	0-16 777 215字节	二进制形式的中等长度文本数据
MEDIUMTEXT	0-16 777 215字节	中等长度文本数据
LOGNGBLOB	0-4 294 967 295字节	二进制形式的极大文本数据
LONGTEXT	0-4 294 967 295字节	极大文本数据

CHAR和VARCHAR类型类似，但它们保存和检索的方式不同。它们的最大长度和是否尾部空格被保留等方面也不同。在存储或检索过程中不进行大小写转换。

BINARY和VARBINARY类类似于CHAR和VARCHAR，不同的是它们包含二进制字符串而不要非二进制字符串。也就是说，它们包含字节字符串而不是字符字符串。这说明它们没有字符集，并且排序和比较基于列值字节的数值值。

BLOB是一个二进制大对象，可以容纳可变数量的数据。有4种BLOB类型：TINYBLOB、BLOB、MEDIUMBLOB和LONGBLOB。它们只是可容纳值的最大长度不同。

有4种TEXT类型：TINYTEXT、TEXT、MEDIUMTEXT和LONGTEXT。这些对应4种BLOB类型，有相同的最大长度和存储需求。

MySQL 创建数据表

创建MySQL数据表需要以下信息：

- 表名
- 表字段名
- 定义每个表字段

语法

以下为创建MySQL数据表的SQL通用语法：

```
CREATE TABLE table_name (column_name column_type);
```

以下例子中我们将在 TUTORIALS 数据库中创建数据表tutorials_tbl：

```
tutorials_tbl(
   tutorial_id INT NOT NULL AUTO_INCREMENT,
   tutorial_title VARCHAR(100) NOT NULL,
   tutorial_author VARCHAR(40) NOT NULL,
   submission_date DATE,
   PRIMARY KEY ( tutorial_id )
);
```

实例解析：

- 如果你不想字段为 **NULL** 可以设置字段的属性为 **NOT NULL**， 在操作数据库时如果输入该字段的数据为**NULL** ，就会报错。
- AUTO_INCREMENT定义列为自增的属性，一般用于主键，数值会自动加1。
- PRIMARY KEY关键字用于定义列为主键。 您可以使用多列来定义主键，列间以逗号分隔。

通过命令提示符创建表

通过 mysql> 命令窗口可以很简单的创建MySQL数据表。你可以使用 SQL 语句 **CREATE TABLE** 来创建数据表。

实例

以下为创建数据表 tutorials_tbl 实例:

```
root@host# mysql -u root -p
Enter password:*******
mysql> use TUTORIALS;
Database changed
mysql> CREATE TABLE tutorials_tbl(
   -> tutorial_id INT NOT NULL AUTO_INCREMENT,
   -> tutorial_title VARCHAR(100) NOT NULL,
   -> tutorial_author VARCHAR(40) NOT NULL,
   -> submission_date DATE,
   -> PRIMARY KEY ( tutorial_id )
   -> );
Query OK, 0 rows affected (0.16 sec)
mysql>
```

注意：MySQL命令终止符为分号 (;) 。

使用PHP脚本创建数据表

你可以使用PHP的 mysql_query() 函数来创建已存在数据库的数据表。

该函数有两个参数，在执行成功时返回 TRUE，否则返回 FALSE。

语法

```
bool mysql_query( sql, connection );
```

参数	描述
sql	必需。规定要发送的 SQL 查询。注释：查询字符串不应以分号结束。
connection	可选。规定 SQL 连接标识符。如果未规定，则使用上一个打开的连接。

实例

以下实例使用了PHP脚本来创建数据表：

```
<html>
<head>
<title>Creating MySQL Tables</title>
</head>
<body>
<?php
$dbhost = 'localhost:3036';
$dbuser = 'root';
$dbpass = 'rootpassword';
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if(! $conn )
{
  die('Could not connect: ' . mysql_error());
}
echo 'Connected successfully<br />';
$sql = "CREATE TABLE tutorials_tbl( ".
       "tutorial_id INT NOT NULL AUTO_INCREMENT, ".
       "tutorial_title VARCHAR(100) NOT NULL, ".
       "tutorial_author VARCHAR(40) NOT NULL, ".
       "submission_date DATE, ".
       "PRIMARY KEY ( tutorial_id )); ";
mysql_select_db( 'TUTORIALS' );
$retval = mysql_query( $sql, $conn );
if(! $retval )
{
  die('Could not create table: ' . mysql_error());
}
echo "Table created successfully\n";
mysql_close($conn);
?>
</body>
</html>
```

MySQL 删除数据表

MySQL中删除数据表是非常容易操作的， 但是你再进行删除表操作时要非常小心，因为执行删除命令后所有数据都会消失。

语法

以下为删除MySQL数据表的通用语法：

```
DROP TABLE table_name ;
```

在命令提示窗口中删除数据表

在mysql>命令提示窗口中删除数据表SQL语句为 **DROP TABLE** ：

实例

以下实例删除了数据表tutorials_tbl:

```
root@host# mysql -u root -p
Enter password:*******
mysql> use TUTORIALS;
Database changed
mysql> DROP TABLE tutorials_tbl
Query OK, 0 rows affected (0.8 sec)
mysql>
```

使用PHP脚本删除数据表

PHP使用 mysql_query 函数来删除 MySQL 数据表。

该函数有两个参数，在执行成功时返回 TRUE，否则返回 FALSE。

h3> 语法

```
bool mysql_query( sql, connection );
```

参数	描述
sql	必需。规定要发送的 SQL 查询。注释：查询字符串不应以分号结束。
connection	可选。规定 SQL 连接标识符。如果未规定，则使用上一个打开的连接。

实例

以下实例使用了PHP脚本删除数据表tutorials_tbl:

```
<html>
<head>
<title>Creating MySQL Tables</title>
</head>
<body>
<?php
$dbhost = 'localhost:3036';
$dbuser = 'root';
$dbpass = 'rootpassword';
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if(! $conn )
{
  die('Could not connect: ' . mysql_error());
}
echo 'Connected successfully<br />';
$sql = "DROP TABLE tutorials_tbl";
mysql_select_db( 'TUTORIALS' );
$retval = mysql_query( $sql, $conn );
if(! $retval )
{
  die('Could not delete table: ' . mysql_error());
}
echo "Table deleted successfully\n";
mysql_close($conn);
?>
</body>
</html>
```

MySQL 插入数据

MySQL 表中使用 **INSERT INTO** SQL语句来插入数据。

你可以通过 mysql> 命令提示窗口中向数据表中插入数据，或者通过PHP脚本来插入数据。

语法

以下为向MySQL数据表插入数据通用的 **INSERT INTO** SQL语法：

```
INSERT INTO table_name ( field1, field2,...fieldN )
                       VALUES
                       ( value1, value2,...valueN );
```

如果数据是字符型，必须使用单引号或者双引号，如："value"。

通过命令提示窗口插入数据

以下我们将使用 SQL **INSERT INTO** 语句向 MySQL 数据表 tutorials_tbl 插入数据

实例

以下实例中我们将想 tutorials_tbl 表插入三条数据:

```
root@host# mysql -u root -p password;
Enter password:*******
mysql> use TUTORIALS;
Database changed
mysql> INSERT INTO tutorials_tbl
     ->(tutorial_title, tutorial_author, submission_date)
     ->VALUES
     ->("Learn PHP", "John Poul", NOW());
Query OK, 1 row affected (0.01 sec)
mysql> INSERT INTO tutorials_tbl
     ->(tutorial_title, tutorial_author, submission_date)
     ->VALUES
     ->("Learn MySQL", "Abdul S", NOW());
Query OK, 1 row affected (0.01 sec)
mysql> INSERT INTO tutorials_tbl
     ->(tutorial_title, tutorial_author, submission_date)
     ->VALUES
     ->("JAVA Tutorial", "Sanjay", '2007-05-06');
Query OK, 1 row affected (0.01 sec)
mysql>
```

注意： 使用箭头标记(->)不是SQL语句的一部分，它仅仅表示一个新行，如果一条SQL语句太长，我们可以通过回车键来创建一个新行来编写SQL语句，SQL语句的命令结束符为分号（;）。

在以上实例中，我们并没有提供 tutorial_id 的数据，因为该字段我们在创建表的时候已经设置它为 AUTO_INCREMENT(自动增加) 属性。 所以，该字段会自动递增而不需要我们去设置。实例中 NOW() 是一个 MySQL 函数，该函数返回日期和时间。

使用**PHP**脚本插入数据

你可以使用PHP 的 mysql_query() 函数来执行 **SQL INSERT INTO**命令来插入数据。

该函数有两个参数，在执行成功时返回 TRUE，否则返回 FALSE。

语法

```
bool mysql_query( sql, connection );
```

参数	描述
sql	必需。规定要发送的 SQL 查询。注释：查询字符串不应以分号结束。
connection	可选。规定 SQL 连接标识符。如果未规定，则使用上一个打开的连接。

实例

以下实例中程序接收用户输入的三个字段数据，并插入数据表中：

```
<html>
<head>
<title>Add New Record in MySQL Database</title>
</head>
<body>
<?php
if(isset($_POST['add']))
{
$dbhost = 'localhost:3036';
$dbuser = 'root';
$dbpass = 'rootpassword';
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if(! $conn )
{
  die('Could not connect: ' . mysql_error());
}

if(! get_magic_quotes_gpc() )
{
   $tutorial_title = addslashes ($_POST['tutorial_title']);
   $tutorial_author = addslashes ($_POST['tutorial_author']);
}
else
{
   $tutorial_title = $_POST['tutorial_title'];
   $tutorial_author = $_POST['tutorial_author'];
}
$submission_date = $_POST['submission_date'];

$sql = "INSERT INTO tutorials_tbl ".
       "(tutorial_title,tutorial_author, submission_date) ".
       "VALUES ".
       "('$tutorial_title','$tutorial_author','$submission_date')";
mysql_select_db('TUTORIALS');
$retval = mysql_query( $sql, $conn );
if(! $retval )
{
  die('Could not enter data: ' . mysql_error());
}
```

```
echo "Entered data successfully\n";
mysql_close($conn);
}
else
{
?>
<form method="post" action="<?php $_PHP_SELF ?>">
<table width="600" border="0" cellspacing="1" cellpadding="2">
<tr>
<td width="250">Tutorial Title</td>
<td>
<input name="tutorial_title" type="text" id="tutorial_title">
</td>
</tr>
<tr>
<td width="250">Tutorial Author</td>
<td>
<input name="tutorial_author" type="text" id="tutorial_author">
</td>
</tr>
<tr>
<td width="250">Submission Date [ yyyy-mm-dd ]</td>
<td>
<input name="submission_date" type="text" id="submission_date">
</td>
</tr>
<tr>
<td width="250"> </td>
<td> </td>
</tr>
<tr>
<td width="250"> </td>
<td>
<input name="add" type="submit" id="add" value="Add Tutorial">
</td>
</tr>
</table>
</form>
<?php
}
?>
</body>
</html>
```

在我们接收用户提交的数据时，为了数据的安全性我们需要使用 get_magic_quotes_gpc() 函数来判断特殊字符的转义是否已经开启。如果这个选项为off（未开启），返回0，那么我们就必须调用addslashes 这个函数来为字符串增加转义。

义。

你也可以添加其他检查数据的方法，比如邮箱格式验证，电话号码验证，是否为整数验证等。

MySQL 查询数据

MySQL 数据库使用SQL SELECT语句来查询数据。

你可以通过 mysql> 命令提示窗口中在数据库中查询数据，或者通过PHP脚本来查询数据。

语法

以下为在MySQL数据库中查询数据通用的 SELECT 语法：

```
SELECT field1, field2,...fieldN table_name1, table_name2...
[WHERE Clause]
[OFFSET M ][LIMIT N]
```

- 查询语句中你可以使用一个或者多个表，表之间使用逗号(,)分割，并使用WHERE语句来设定查询条件。
- SELECT 命令可以读取一条或者多条记录。
- 你可以使用星号（*）来代替其他字段，SELECT语句会返回表的所有字段数据
- 你可以使用 WHERE 语句来包含任何条件。
- 你可以通过OFFSET指定SELECT语句开始查询的数据偏移量。默认情况下偏移量为0。
- 你可以使用 LIMIT 属性来设定返回的记录数。

通过命令提示符获取数据

以下实例我们将通过 SQL SELECT 命令来获取 MySQL 数据表 tutorials_tbl 的数据：

实例

以下实例将返回数据表tutorials_tbl的所有记录:

```
root@host# mysql -u root -p password;
Enter password:*******
mysql> use TUTORIALS;
Database changed
mysql> SELECT * from tutorials_tbl
+-------------+----------------+-----------------+-----------------
| tutorial_id | tutorial_title | tutorial_author | submission_date
+-------------+----------------+-----------------+-----------------
|           1 | Learn PHP      | John Poul       | 2007-05-21
|           2 | Learn MySQL    | Abdul S         | 2007-05-21
|           3 | JAVA Tutorial  | Sanjay          | 2007-05-21
+-------------+----------------+-----------------+-----------------
3 rows in set (0.01 sec)

mysql>
```

使用PHP脚本来获取数据

使用PHP函数的mysql_query()及SQL SELECT命令来获取数据。

该函数用于执行SQL命令，然后通过 PHP 函数 mysql_fetch_array() 来使用或输出所有查询的数据。

mysql_fetch_array() 函数从结果集中取得一行作为关联数组，或数字数组，或二者兼有 返回根据从结果集取得的行生成的数组，如果没有更多行则返回 false。

以下实例为从数据表 tutorials_tbl 中读取所有记录。

实例

尝试以下实例来显示数据表 tutorials_tbl 的所有记录。

```
<?php
$dbhost = 'localhost:3036';
$dbuser = 'root';
$dbpass = 'rootpassword';
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if(! $conn )
{
  die('Could not connect: ' . mysql_error());
}
$sql = 'SELECT tutorial_id, tutorial_title,
               tutorial_author, submission_date
        FROM tutorials_tbl';

mysql_select_db('TUTORIALS');
$retval = mysql_query( $sql, $conn );
if(! $retval )
{
  die('Could not get data: ' . mysql_error());
}
while($row = mysql_fetch_array($retval, MYSQL_ASSOC))
{
    echo "Tutorial ID :{$row['tutorial_id']}  <br> ".
         "Title: {$row['tutorial_title']} <br> ".
         "Author: {$row['tutorial_author']} <br> ".
         "Submission Date : {$row['submission_date']} <br> ".
         "--------------------------------<br>";
}
echo "Fetched data successfully\n";
mysql_close($conn);
?>
```

以上实例中，读取的每行记录赋值给变量$row，然后再打印出每个值。

注意：记住如果你需要在字符串中使用变量，请将变量置于花括号。

在上面的例子中，PHP mysql_fetch_array()函数第二个参数为MYSQL_ASSOC，设置该参数查询结果返回关联数组，你可以使用字段名称来作为数组的索引。

PHP提供了另外一个函数mysql_fetch_assoc(), 该函数从结果集中取得一行作为关联数组。 返回根据从结果集取得的行生成的关联数组，如果没有更多行，则返回false。

实例

尝试以下实例，该实例使用了mysql_fetch_assoc()函数来输出数据表tutorial_tbl的所有记录：

```
<?php
$dbhost = 'localhost:3036';
$dbuser = 'root';
$dbpass = 'rootpassword';
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if(! $conn )
{
  die('Could not connect: ' . mysql_error());
}
$sql = 'SELECT tutorial_id, tutorial_title,
               tutorial_author, submission_date
        FROM tutorials_tbl';

mysql_select_db('TUTORIALS');
$retval = mysql_query( $sql, $conn );
if(! $retval )
{
  die('Could not get data: ' . mysql_error());
}
while($row = mysql_fetch_assoc($retval))
{
    echo "Tutorial ID :{$row['tutorial_id']}  <br> ".
         "Title: {$row['tutorial_title']} <br> ".
         "Author: {$row['tutorial_author']} <br> ".
         "Submission Date : {$row['submission_date']} <br> ".
         "--------------------------------<br>";
}
echo "Fetched data successfully\n";
mysql_close($conn);
?>
```

你也可以使用常量 MYSQL_NUM 作为PHP mysql_fetch_array()函数的第二个参数，返回数字数组。

实例

以下实例使用MYSQL_NUM参数显示数据表tutorials_tbl的所有记录:

```
<?php
$dbhost = 'localhost:3036';
$dbuser = 'root';
$dbpass = 'rootpassword';
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if(! $conn )
{
  die('Could not connect: ' . mysql_error());
}
$sql = 'SELECT tutorial_id, tutorial_title,
               tutorial_author, submission_date
        FROM tutorials_tbl';

mysql_select_db('TUTORIALS');
$retval = mysql_query( $sql, $conn );
if(! $retval )
{
  die('Could not get data: ' . mysql_error());
}
while($row = mysql_fetch_array($retval, MYSQL_NUM))
{
    echo "Tutorial ID :{$row[0]}  <br> ".
         "Title: {$row[1]} <br> ".
         "Author: {$row[2]} <br> ".
         "Submission Date : {$row[3]} <br> ".
         "--------------------------------<br>";
}
echo "Fetched data successfully\n";
mysql_close($conn);
?>
```

以上三个实例输出结果都一样。

内存释放

在我们执行完SELECT语句后，释放游标内存是一个很好的习惯。 。可以通过PHP函数mysql_free_result()来实现内存的释放。

以下实例演示了该函数的使用方法。

实例

尝试以下实例:

```
<?php
$dbhost = 'localhost:3036';
$dbuser = 'root';
$dbpass = 'rootpassword';
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if(! $conn )
{
  die('Could not connect: ' . mysql_error());
}
$sql = 'SELECT tutorial_id, tutorial_title,
               tutorial_author, submission_date
        FROM tutorials_tbl';

mysql_select_db('TUTORIALS');
$retval = mysql_query( $sql, $conn );
if(! $retval )
{
  die('Could not get data: ' . mysql_error());
}
while($row = mysql_fetch_array($retval, MYSQL_NUM))
{
    echo "Tutorial ID :{$row[0]}  <br> ".
         "Title: {$row[1]} <br> ".
         "Author: {$row[2]} <br> ".
         "Submission Date : {$row[3]} <br> ".
         "--------------------------------<br>";
}
mysql_free_result($retval);
echo "Fetched data successfully\n";
mysql_close($conn);
?>
```

MySQL where 子句

我们知道从MySQL表中使用SQL SELECT 语句来读取数据。

如需有条件地从表中选取数据，可将 WHERE 子句添加到 SELECT 语句中。

语法

以下是SQL SELECT 语句使用 WHERE 子句从数据表中读取数据的通用语法：

```
SELECT field1, field2,...fieldN table_name1, table_name2...
[WHERE condition1 [AND [OR]] condition2.....
```

- 查询语句中你可以使用一个或者多个表，表之间使用逗号(,)分割，并使用WHERE语句来设定查询条件。
- 你可以在WHERE子句中指定任何条件。
- 你可以使用AND或者OR指定一个或多个条件。
- WHERE子句也可以运用于SQL的 DELETE 或者 UPDATE 命令。
- WHERE 子句类似于程序语言中的if条件，根据 MySQL 表中的字段值来读取指定的数据。

以下为操作符列表，可用于 WHERE 子句中。

下表中实例假定 A为10 B为20

操作符	描述	实例
=	等号，检测两个值是否相等，如果相等返回true	(A = B) 返回false。
<>, !=	不等于，检测两个值是否相等，如果不相等返回true	(A != B) 返回 true。
>	大于号，检测左边的值是否大于右边的值, 如果左边的值大于右边的值返回true	(A > B) 返回false。
<	小于号，检测左边的值是否小于右边的值, 如果左边的值小于右边的值返回true	(A < B) 返回 true。
>=	大于等于号，检测左边的值是否大于或等于右边的值, 如果左边的值大于或等于右边的值返回true	(A >= B) 返回 false。
<=	小于等于号，检测左边的值是否小于于或等于右边的值, 如果左边的值小于或等于右边的值返回true	(A <= B) 返回 true。

如果我们想再MySQL数据表中读取指定的数据，WHERE 子句是非常有用的。

使用主键来作为 WHERE 子句的条件查询是非常快速的。

如果给定的条件在表中没有任何匹配的记录，那么查询不会返回任何数据。

从命令提示符中读取数据

我们将在SQL SELECT语句使用WHERE子句来读取MySQL数据表 tutorials_tbl 中的数据：

实例

以下实例将读取 tutorials_tbl 表中 tutorial_author 字段值为 Sanjay 的所有记录：

```
root@host# mysql -u root -p password;
Enter password:*******
mysql> use TUTORIALS;
Database changed
mysql> SELECT * from tutorials_tbl WHERE tutorial_author='Sanjay';
+-------------+----------------+-----------------+-----------------
| tutorial_id | tutorial_title | tutorial_author | submission_date
+-------------+----------------+-----------------+-----------------
|           3 | JAVA Tutorial  | Sanjay          | 2007-05-21
+-------------+----------------+-----------------+-----------------
1 rows in set (0.01 sec)

mysql>
```

除非你使用 LIKE 来比较字符串，否则MySQL的WHERE子句的字符串比较是不区分大小写的。 你可以使用 BINARY 关键字来设定WHERE子句的字符串比较是区分大小写的。

如下实例

```
root@host# mysql -u root -p password;
Enter password:*******
mysql> use TUTORIALS;
Database changed
mysql> SELECT * from tutorials_tbl \
          WHERE BINARY tutorial_author='sanjay';
Empty set (0.02 sec)

mysql>
```

使用PHP脚本读取数据

你可以使用PHP函数的mysql_query()及相同的SQL SELECT 带上 WHERE 子句的命令来获取数据。

该函数用于执行SQL命令，然后通过 PHP 函数 mysql_fetch_array() 来输出所有查询的数据。

实例

以下实例将从 tutorials_tbl 表中返回使用 tutorial_author 字段值为 Sanjay 的记录：

```
<?php
$dbhost = 'localhost:3036';
$dbuser = 'root';
$dbpass = 'rootpassword';
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if(! $conn )
{
  die('Could not connect: ' . mysql_error());
}
$sql = 'SELECT tutorial_id, tutorial_title,
               tutorial_author, submission_date
        FROM tutorials_tbl
        WHERE tutorial_author="Sanjay"';

mysql_select_db('TUTORIALS');
$retval = mysql_query( $sql, $conn );
if(! $retval )
{
  die('Could not get data: ' . mysql_error());
}
while($row = mysql_fetch_array($retval, MYSQL_ASSOC))
{
    echo "Tutorial ID :{$row['tutorial_id']}  <br> ".
         "Title: {$row['tutorial_title']} <br> ".
         "Author: {$row['tutorial_author']} <br> ".
         "Submission Date : {$row['submission_date']} <br> ".
         "--------------------------------<br>";
}
echo "Fetched data successfully\n";
mysql_close($conn);
?>
```

MySQL UPDATE 查询

如果我们需要修改或更新MySQL中的数据，我们可以使用 SQL UPDATE 命令来操作。.

语法

以下是 UPDATE 命令修改 MySQL 数据表数据的通用SQL语法：

```
UPDATE table_name SET field1=new-value1, field2=new-value2
[WHERE Clause]
```

- 你可以同时更新一个或多个字段。
- 你可以在 WHERE 子句中指定任何条件。
- 你可以在一个单独表中同时更新数据。

当你需要更新数据表中指定行的数据时 WHERE 子句是非常有用的。

通过命令提示符更新数据

以下我们将在 SQL UPDATE 命令使用 WHERE子句来更新tutorials_tbl表中指定的数据：

实例

以下实例将更新数据表中 tutorial_id 为 3 的 tutorial_title 字段值：

```
root@host# mysql -u root -p password;
Enter password:*******
mysql> use TUTORIALS;
Database changed
mysql> UPDATE tutorials_tbl
    -> SET tutorial_title='Learning JAVA'
    -> WHERE tutorial_id=3;
Query OK, 1 row affected (0.04 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql>
```

使用PHP脚本更新数据

PHP中使用函数mysql_query()来执行SQL语句，你可以在SQL UPDATE语句中使用或者不适用WHERE子句。

该函数与在mysql>命令提示符中执行SQL语句的效果是一样的。

实例

以下实例将更新 tutorial_id 为3的 tutorial_title 字段的数据。

```
<?php
$dbhost = 'localhost:3036';
$dbuser = 'root';
$dbpass = 'rootpassword';
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if(! $conn )
{
  die('Could not connect: ' . mysql_error());
}
$sql = 'UPDATE tutorials_tbl
        SET tutorial_title="Learning JAVA"
        WHERE tutorial_id=3';

mysql_select_db('TUTORIALS');
$retval = mysql_query( $sql, $conn );
if(! $retval )
{
  die('Could not update data: ' . mysql_error());
}
echo "Updated data successfully\n";
mysql_close($conn);
?>
```

MySQL DELETE 语句

你可以使用 SQL 的 DELETE FROM 命令来删除 MySQL 数据表中的记录。

你可以在mysql>命令提示符或PHP脚本中执行该命令。

语法

以下是SQL DELETE 语句从MySQL数据表中删除数据的通用语法：

```
DELETE FROM table_name [WHERE Clause]
```

- 如果没有指定 WHERE 子句，MySQL表中的所有记录将被删除。
- 你可以在 WHERE 子句中指定任何条件
- 您可以在单个表中一次性删除记录。

当你想删除数据表中指定的记录时 WHERE 子句是非常有用的。

从命令行中删除数据

这里我们将在 SQL DELETE 命令中使用 WHERE 子句来删除MySQL数据表 tutorials_tbl所选的数据。

实例

以下实例将删除 tutorial_tbl 表中 tutorial_id 为3 的记录：

```
root@host# mysql -u root -p password;
Enter password:*******
mysql> use TUTORIALS;
Database changed
mysql> DELETE FROM tutorials_tbl WHERE tutorial_id=3;
Query OK, 1 row affected (0.23 sec)

mysql>
```

使用 PHP 脚本删除数据

PHP使用 mysql_query() 函数来执行SQL语句， 你可以在SQL DELETE命令中使用或不使用 WHERE 子句。

该函数与 mysql>命令符执行SQL命令的效果是一样的。

实例

以下PHP实例将删除tutorial_tbl表中tutorial_id为3的记录:

```
<?php
$dbhost = 'localhost:3036';
$dbuser = 'root';
$dbpass = 'rootpassword';
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if(! $conn )
{
  die('Could not connect: ' . mysql_error());
}
$sql = 'DELETE FROM tutorials_tbl
        WHERE tutorial_id=3';

mysql_select_db('TUTORIALS');
$retval = mysql_query( $sql, $conn );
if(! $retval )
{
  die('Could not delete data: ' . mysql_error());
}
echo "Deleted data successfully\n";
mysql_close($conn);
?>
```

MySQL LIKE 子句

我们知道在MySQL中使用 SQL SELECT 命令来读取数据， 同时我们可以在 SELECT 语句中使用 WHERE 子句来获取指定的记录。

WHERE 子句中可以使用等号 (=) 来设定获取数据的条件，如 "tutorial_author = 'Sanjay'"。

但是有时候我们需要获取 tutorial_author 字段含有 "jay" 字符的所有记录，这时我们就需要在 WHERE 子句中使用 SQL LIKE 子句。

SQL LIKE 子句中使用百分号(%)字符来表示任意字符，类似于UNIX或正则表达式中的星号 (*)。

如果没有使用百分号(%), LIKE 子句与等号（=）的效果是一样的。

语法

以下是SQL SELECT 语句使用 LIKE 子句从数据表中读取数据的通用语法：

```
SELECT field1, field2,...fieldN table_name1, table_name2...
WHERE field1 LIKE condition1 [AND [OR]] filed2 = 'somevalue'
```

- 你可以在WHERE子句中指定任何条件。
- 你可以在WHERE子句中使用LIKE子句。
- 你可以使用LIKE子句代替等号(=)。
- LIKE 通常与 % 一同使用，类似于一个元字符的搜索。
- 你可以使用AND或者OR指定一个或多个条件。
- 你可以在 DELETE 或 UPDATE 命令中使用 WHERE...LIKE 子句来指定条件。

在命令提示符中使用 LIKE 子句

以下我们将在 SQL SELECT 命令中使用 WHERE...LIKE 子句来从MySQL数据表 tutorials_tbl 中读取数据。

实例

以下是我们将tutorials_tbl表中获取tutorial_author字段中以"jay"为结尾的的所有记录：

```
root@host# mysql -u root -p password;
Enter password:*******
mysql> use TUTORIALS;
Database changed
mysql> SELECT * from tutorials_tbl
    -> WHERE tutorial_author LIKE '%jay';
+-------------+----------------+-----------------+----------------
| tutorial_id | tutorial_title | tutorial_author | submission_date
+-------------+----------------+-----------------+----------------
|           3 | JAVA Tutorial  | Sanjay          | 2007-05-21
+-------------+----------------+-----------------+----------------
1 rows in set (0.01 sec)

mysql>
```

在PHP脚本中使用 LIKE 子句

你可以使用PHP函数的mysql_query()及相同的SQL SELECT 带上 WHERE...LIKE 子句的命令来获取数据。

该函数用于执行SQL命令，然后通过 PHP 函数 mysql_fetch_array() 来输出所有查询的数据。

但是如果是DELETE或者UPDATE中使用 WHERE...LIKE 子句的SQL语句，则无需使用mysql_fetch_array() 函数。

实例

以下是我们使用PHP脚本在tutorials_tbl表中读取tutorial_author字段中以"jay"为结尾的的所有记录：

```
<?php
$dbhost = 'localhost:3036';
$dbuser = 'root';
$dbpass = 'rootpassword';
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if(! $conn )
{
  die('Could not connect: ' . mysql_error());
}
$sql = 'SELECT tutorial_id, tutorial_title,
               tutorial_author, submission_date
        FROM tutorials_tbl
        WHERE tutorial_author LIKE "%jay%"';

mysql_select_db('TUTORIALS');
$retval = mysql_query( $sql, $conn );
if(! $retval )
{
  die('Could not get data: ' . mysql_error());
}
while($row = mysql_fetch_array($retval, MYSQL_ASSOC))
{
    echo "Tutorial ID :{$row['tutorial_id']}  <br> ".
         "Title: {$row['tutorial_title']} <br> ".
         "Author: {$row['tutorial_author']} <br> ".
         "Submission Date : {$row['submission_date']} <br> ".
         "--------------------------------<br>";
}
echo "Fetched data successfully\n";
mysql_close($conn);
?>
```

MySQL 排序

我们知道从MySQL表中使用SQL SELECT 语句来读取数据。

如果我们需要对读取的数据进行排序，我们就可以使用MySQL的 ORDER BY 子句来设定你想按哪个字段哪中方式来进行排序，再返回搜索结果。

语法

以下是SQL SELECT 语句使用 ORDER BY 子句将查询数据排序后再返回数据：

```
SELECT field1, field2,...fieldN table_name1, table_name2...
ORDER BY field1, [field2...] [ASC [DESC]]
```

- 你可以使用任何字段来作为排序的条件，从而返回排序后的查询结果。
- 你可以设定多个字段来排序。
- 你可以使用 ASC 或 DESC 关键字来设置查询结果是按升序或降序排列。 默认情况下，它是按升排列。
- 你可以添加 WHERE...LIKE 子句来设置条件。

在命令提示符中使用 ORDER BY 子句

以下将在 SQL SELECT 语句中使用 ORDER BY 子句来读取MySQL 数据表 tutorials_tbl 中的数据：

实例

尝试以下实例，结果将按升序排列

```
root@host# mysql -u root -p password;
Enter password:*******
mysql> use TUTORIALS;
Database changed
mysql> SELECT * from tutorials_tbl ORDER BY tutorial_author ASC
+-------------+----------------+-----------------+-----------------
| tutorial_id | tutorial_title | tutorial_author | submission_date
+-------------+----------------+-----------------+-----------------
|           2 | Learn MySQL    | Abdul S         | 2007-05-24
|           1 | Learn PHP      | John Poul       | 2007-05-24
|           3 | JAVA Tutorial  | Sanjay          | 2007-05-06
+-------------+----------------+-----------------+-----------------
3 rows in set (0.42 sec)

mysql>
```

读取 tutorials_tbl 表中所有数据并按 tutorial_author 字段的升序排列。

在PHP脚本中使用 ORDER BY 子句

你可以使用PHP函数的mysql_query()及相同的SQL SELECT 带上 ORDER BY 子句的命令来获取数据。 该函数用于执行SQL命令，然后通过 PHP 函数 mysql_fetch_array() 来输出所有查询的数据。

实例

尝试以下实例，查询后的数据按 tutorial_author 字段的降序排列后返回。

```
<?php
$dbhost = 'localhost:3036';
$dbuser = 'root';
$dbpass = 'rootpassword';
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if(! $conn )
{
  die('Could not connect: ' . mysql_error());
}
$sql = 'SELECT tutorial_id, tutorial_title,
               tutorial_author, submission_date
        FROM tutorials_tbl
        ORDER BY  tutorial_author DESC';

mysql_select_db('TUTORIALS');
$retval = mysql_query( $sql, $conn );
if(! $retval )
{
  die('Could not get data: ' . mysql_error());
}
while($row = mysql_fetch_array($retval, MYSQL_ASSOC))
{
    echo "Tutorial ID :{$row['tutorial_id']}  <br> ".
         "Title: {$row['tutorial_title']} <br> ".
         "Author: {$row['tutorial_author']} <br> ".
         "Submission Date : {$row['submission_date']} <br> ".
         "--------------------------------<br>";
}
echo "Fetched data successfully\n";
mysql_close($conn);
?>
```

Mysql Join的使用

在前几章节中，我们已经学会了如果在一张表中读取数据，这是相对简单的，但是在真正的应用中经常需要从多个数据表中读取数据。

本章节我们将向大家介绍如何使用MySQL 的 JOIN 在两个或多个表中查询数据。

你可以在SELECT, UPDATE 和 DELETE 语句中使用Mysql 的 join 来联合多表查询。

以下我们将演示MySQL LEFT JOIN 和 JOIN 的使用的不同之处。

在命令提示符中使用JOIN

我们在TUTORIALS数据库中有两张表 tcount_tbl 和 tutorials_tbl。两张数据表数据如下：

实例

尝试以下实例：

```
root@host# mysql -u root -p password;
Enter password:*******
mysql> use TUTORIALS;
Database changed
mysql> SELECT * FROM tcount_tbl;
+-----------------+----------------+
| tutorial_author | tutorial_count |
+-----------------+----------------+
| mahran          |             20 |
| mahnaz          |           NULL |
| Jen             |           NULL |
| Gill            |             20 |
| John Poul       |              1 |
| Sanjay          |              1 |
+-----------------+----------------+
6 rows in set (0.01 sec)
mysql> SELECT * from tutorials_tbl;
+-------------+----------------+-----------------+-----------------
| tutorial_id | tutorial_title | tutorial_author | submission_date
+-------------+----------------+-----------------+-----------------
|           1 | Learn PHP      | John Poul       | 2007-05-24
|           2 | Learn MySQL    | Abdul S         | 2007-05-24
|           3 | JAVA Tutorial  | Sanjay          | 2007-05-06
+-------------+----------------+-----------------+-----------------
3 rows in set (0.00 sec)
mysql>
```

接下来我们就使用MySQL的JOIN来连接以上两张表来读取tutorials_tbl表中所有tutorial_author字段在tcount_tbl表对应的tutorial_count字段值：

```
mysql> SELECT a.tutorial_id, a.tutorial_author, b.tutorial_count
    -> FROM tutorials_tbl a, tcount_tbl b
    -> WHERE a.tutorial_author = b.tutorial_author;
+-------------+-----------------+----------------+
| tutorial_id | tutorial_author | tutorial_count |
+-------------+-----------------+----------------+
|           1 | John Poul       |              1 |
|           3 | Sanjay          |              1 |
+-------------+-----------------+----------------+
2 rows in set (0.01 sec)
mysql>
```

在PHP脚本中使用JOIN

PHP 中使用mysql_query()函数来执行SQL语句，你可以使用以上的相同的SQL语句作为mysql_query()函数的参数。

尝试如下实例:

```
<?php
$dbhost = 'localhost:3036';
$dbuser = 'root';
$dbpass = 'rootpassword';
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if(! $conn )
{
  die('Could not connect: ' . mysql_error());
}
$sql = 'SELECT a.tutorial_id, a.tutorial_author, b.tutorial_count
        FROM tutorials_tbl a, tcount_tbl b
        WHERE a.tutorial_author = b.tutorial_author';

mysql_select_db('TUTORIALS');
$retval = mysql_query( $sql, $conn );
if(! $retval )
{
  die('Could not get data: ' . mysql_error());
}
while($row = mysql_fetch_array($retval, MYSQL_ASSOC))
{
    echo "Author:{$row['tutorial_author']}  <br> ".
         "Count: {$row['tutorial_count']} <br> ".
         "Tutorial ID: {$row['tutorial_id']} <br> ".
         "--------------------------------<br>";
}
echo "Fetched data successfully\n";
mysql_close($conn);
?>
```

MySQL LEFT JOIN

MySQL left join 与 join 有所不同。 MySQL LEFT JOIN 会读取左边数据表的全部数据，即便右边表无对应数据。

实例

尝试以下实例，理解MySQL LEFT JOIN的应用：

```
root@host# mysql -u root -p password;
Enter password:*******
mysql> use TUTORIALS;
Database changed
mysql> SELECT a.tutorial_id, a.tutorial_author, b.tutorial_count
    -> FROM tutorials_tbl a LEFT JOIN tcount_tbl b
    -> ON a.tutorial_author = b.tutorial_author;
+-------------+-----------------+----------------+
| tutorial_id | tutorial_author | tutorial_count |
+-------------+-----------------+----------------+
|           1 | John Poul       |              1 |
|           2 | Abdul S         |           NULL |
|           3 | Sanjay          |              1 |
+-------------+-----------------+----------------+
3 rows in set (0.02 sec)
```

以上实例中使用了LEFT JOIN，该语句会读取左边的数据表tutorials_tbl的所有选取的字段数据，即便在右侧表tcount_tbl中没有对应的tutorial_author字段值。

MySQL NULL 值处理

我们已经知道MySQL使用 SQL SELECT 命令及 WHERE 子句来读取数据表中的数据,但是当提供的查询条件字段为 NULL 时，该命令可能就无法正常工作。

为了处理这种情况，MySQL提供了三大运算符:

- **IS NULL:** 当列的值是NULL,此运算符返回true。
- **IS NOT NULL:** 当列的值不为NULL, 运算符返回true。
- **<=>:** 比较操作符（不同于=运算符），当比较的的两个值为NULL时返回true。

关于 NULL 的条件比较运算是比较特殊的。你不能使用 = NULL 或 != NULL 在列中查找 NULL 值 。

在MySQL中，NULL值与任何其它值的比较（即使是NULL）永远返回false，即 NULL = NULL 返回false 。

MySQL中处理NULL使用IS NULL和IS NOT NULL运算符。

在命令提示符中使用 NULL 值

以下实例中假设数据库 TUTORIALS 中的表 tcount_tbl 含有两列 tutorial_author 和 tutorial_count, tutorial_count 中设置插入NULL值。

实例

尝试以下实例:

```
root@host# mysql -u root -p password;
Enter password:*******
mysql> use TUTORIALS;
Database changed
mysql> create table tcount_tbl
    -> (
    -> tutorial_author varchar(40) NOT NULL,
    -> tutorial_count  INT
    -> );
Query OK, 0 rows affected (0.05 sec)
mysql> INSERT INTO tcount_tbl
    -> (tutorial_author, tutorial_count) values ('mahran', 20);
mysql> INSERT INTO tcount_tbl
    -> (tutorial_author, tutorial_count) values ('mahnaz', NULL);
mysql> INSERT INTO tcount_tbl
    -> (tutorial_author, tutorial_count) values ('Jen', NULL);
mysql> INSERT INTO tcount_tbl
    -> (tutorial_author, tutorial_count) values ('Gill', 20);

mysql> SELECT * from tcount_tbl;
+-----------------+----------------+
| tutorial_author | tutorial_count |
+-----------------+----------------+
| mahran          |             20 |
| mahnaz          |           NULL |
| Jen             |           NULL |
| Gill            |             20 |
+-----------------+----------------+
4 rows in set (0.00 sec)

mysql>
```

以下实例中你可以看到 = 和 != 运算符是不起作用的：

```
mysql> SELECT * FROM tcount_tbl WHERE tutorial_count = NULL;
Empty set (0.00 sec)
mysql> SELECT * FROM tcount_tbl WHERE tutorial_count != NULL;
Empty set (0.01 sec)
```

查找数据表中 tutorial_count 列是否为 NULL，必须使用IS NULL和IS NOT NULL，如下实例：

```
mysql> SELECT * FROM tcount_tbl
    -> WHERE tutorial_count IS NULL;
+-----------------+----------------+
| tutorial_author | tutorial_count |
+-----------------+----------------+
| mahnaz          |           NULL |
| Jen             |           NULL |
+-----------------+----------------+
2 rows in set (0.00 sec)
mysql> SELECT * from tcount_tbl
    -> WHERE tutorial_count IS NOT NULL;
+-----------------+----------------+
| tutorial_author | tutorial_count |
+-----------------+----------------+
| mahran          |             20 |
| Gill            |             20 |
+-----------------+----------------+
2 rows in set (0.00 sec)
```

使用PHP脚本处理 NULL 值

PHP脚本中你可以在 if...else 语句来处理变量是否为空，并生成相应的条件语句。

以下实例中PHP设置了$tutorial_count变量，然后使用该变量与数据表中的tutorial_count 字段进行比较：

```
<?php
$dbhost = 'localhost:3036';
$dbuser = 'root';
$dbpass = 'rootpassword';
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if(! $conn )
{
  die('Could not connect: ' . mysql_error());
}
if( isset($tutorial_count ))
{
   $sql = 'SELECT tutorial_author, tutorial_count
           FROM  tcount_tbl
           WHERE tutorial_count = $tutorial_count';
}
else
{
   $sql = 'SELECT tutorial_author, tutorial_count
           FROM  tcount_tbl
           WHERE tutorial_count IS $tutorial_count';
}

mysql_select_db('TUTORIALS');
$retval = mysql_query( $sql, $conn );
if(! $retval )
{
  die('Could not get data: ' . mysql_error());
}
while($row = mysql_fetch_array($retval, MYSQL_ASSOC))
{
    echo "Author:{$row['tutorial_author']}  <br> ".
         "Count: {$row['tutorial_count']} <br> ".
         "--------------------------------<br>";
}
echo "Fetched data successfully\n";
mysql_close($conn);
?>
```

MySQL 正则表达式

在前面的章节我们已经了解到MySQL可以通过 **LIKE ...%** 来进行模糊匹配。

MySQL 同样也支持其他正则表达式的匹配， MySQL中使用 REGEXP 操作符来进行正则表达式匹配。

如果您了解PHP或Perl，那么操作起来就非常简单，因为MySQL的正则表达式匹配与这些脚本的类似。

下表中的正则模式可应用于 REGEXP 操作符中。

模式	描述
^	匹配输入字符串的开始位置。如果设置了 RegExp 对象的 Multiline 属性，^ 也匹配 '\n' 或 '\r' 之后的位置。
$	匹配输入字符串的结束位置。如果设置了RegExp 对象的 Multiline 属性，$ 也匹配 '\n' 或 '\r' 之前的位置。
.	匹配除 "\n" 之外的任何单个字符。要匹配包括 '\n' 在内的任何字符，请使用象 '[.\n]' 的模式。
[...]	字符集合。匹配所包含的任意一个字符。例如， '[abc]' 可以匹配 "plain" 中的 'a'。
[^...]	负值字符集合。匹配未包含的任意字符。例如， '[^abc]' 可以匹配 "plain" 中的'p'。
p1|p2|p3	匹配 p1 或 p2 或 p3。例如，'z|food' 能匹配 "z" 或 "food"。'(z|f)ood' 则匹配 "zood" 或 "food"。
*	匹配前面的子表达式零次或多次。例如，zo 能匹配 "z" 以及 "zoo"。 等价于{0,}。
+	匹配前面的子表达式一次或多次。例如，'zo+' 能匹配 "zo" 以及 "zoo"，但不能匹配 "z"。+ 等价于 {1,}。
{n}	n 是一个非负整数。匹配确定的 n 次。例如，'o{2}' 不能匹配 "Bob" 中的 'o'，但是能匹配 "food" 中的两个 o。
{n,m}	m 和 n 均为非负整数，其中n <= m。最少匹配 n 次且最多匹配 m 次。

实例

了解以上的正则需求后，我们就可以更加自己的需求来编写带有正则表达式的SQL语句。以下我们将列出几个小实例(表名：person_tbl)来加深我们的理解：

查找name字段中以'st'为开头的所有数据：

```
mysql> SELECT name FROM person_tbl WHERE name REGEXP '^st';
```

查找name字段中以'ok'为结尾的所有数据：

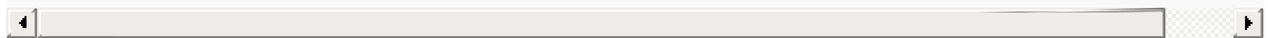
```
mysql> SELECT name FROM person_tbl WHERE name REGEXP 'ok$';
```

查找name字段中包含'mar'字符串的所有数据：

```
mysql> SELECT name FROM person_tbl WHERE name REGEXP 'mar';
```

查找name字段中以元音字符开头且以'ok'字符串结尾的所有数据：

```
mysql> SELECT name FROM person_tbl WHERE name REGEXP '^[aeiou]|ok$
```



MySQL 事务

MySQL 事务主要用于处理操作量大，复杂度高的数据。比如说，在人员管理系统中，你删除一个人员，你即需要删除人员的基本资料，也要删除和该人员相关的信息，如信箱，文章等等，这样，这些数据库操作语句就构成一个事务！

- 在MySQL中只有使用了Innodb数据库引擎的数据库或表才支持事务
- 事务处理可以用来维护数据库的完整性，保证成批的SQL语句要么全部执行，要么全部不执行
- 事务用来管理insert,update,delete语句

一般来说，事务是必须满足4个条件（ACID）： Atomicity（原子性）、Consistency（稳定性）、Isolation（隔离性）、Durability（可靠性）

- 1、事务的原子性：一组事务，要么成功；要么撤回。
- 2、稳定性 ： 有非法数据（外键约束之类），事务撤回。
- 3、隔离性：事务独立运行。一个事务处理后的结果，影响了其他事务，那么其他事务会撤回。事务的100%隔离，需要牺牲速度。
- 4、可靠性：软、硬件崩溃后，InnoDB数据表驱动会利用日志文件重构修改。可靠性和高速度不可兼得， innodb_flush_log_at_trx_commit选项 决定什么时候吧事务保存到日志里。

在Mysql控制台使用事务来操作

1，开始一个事务

```
start transaction
```

2, 做保存点

```
save point 保存点名称
```

3, 操作

4，可以回滚，可以提交，没有问题，就提交，有问题就回滚。

PHP中使用事务实例

```
<?php
$handler=mysql_connect("localhost","root","password");
mysql_select_db("task");
mysql_query("SET AUTOCOMMIT=0");//设置为不自动提交，因为MYSQL默认立即执行
mysql_query("BEGIN");//开始事务定义
if(!mysql_query("insert into trans (id) values('2')"))
{
mysql_query("ROOLBACK");//判断当执行失败时回滚
}
if(!mysql_query("insert into trans (id) values('4')"))
{
mysql_query("ROOLBACK");//判断执行失败回滚
}
mysql_query("COMMIT");//执行事务
mysql_close($handler);
?>
```

MySQL ALTER命令

当我们需要修改数据表名或者修改数据表字段时，就需要使用到MySQL ALTER命令。

开始本章教程前让我们先创建一张表，表名为：testalter_tbl。

```
root@host# mysql -u root -p password;
Enter password:*******
mysql> use TUTORIALS;
Database changed
mysql> create table testalter_tbl
    -> (
    -> i INT,
    -> c CHAR(1)
    -> );
Query OK, 0 rows affected (0.05 sec)
mysql> SHOW COLUMNS FROM testalter_tbl;
+-------+---------+------+-----+---------+-------+
| Field | Type    | Null | Key | Default | Extra |
+-------+---------+------+-----+---------+-------+
| i     | int(11) | YES  |     | NULL    |       |
| c     | char(1) | YES  |     | NULL    |       |
+-------+---------+------+-----+---------+-------+
2 rows in set (0.00 sec)
```

删除，添加或修改表字段

如下命令使用了 ALTER 命令及 DROP 子句来删除以上创建表的 i 字段：

```
mysql> ALTER TABLE testalter_tbl  DROP i;
```

如果数据表中只剩余一个字段则无法使用DROP来删除字段。

MySQL 中使用 ADD 子句来想数据表中添加列，如下实例在表 testalter_tbl 中添加 i 字段，并定义数据类型:

```
mysql> ALTER TABLE testalter_tbl ADD i INT;
```

执行以上命令后，i 字段会自动添加到数据表字段的末尾。

```
mysql> SHOW COLUMNS FROM testalter_tbl;
+-------+---------+------+-----+---------+-------+
| Field | Type    | Null | Key | Default | Extra |
+-------+---------+------+-----+---------+-------+
| c     | char(1) | YES  |     | NULL    |       |
| i     | int(11) | YES  |     | NULL    |       |
+-------+---------+------+-----+---------+-------+
2 rows in set (0.00 sec)
```

如果你需要指定新增字段的位置，可以使用MySQL提供的关键字 FIRST (设定位第一列)， AFTER 字段名（设定位于某个字段之后）。

尝试以下 ALTER TABLE 语句, 在执行成功后，使用 SHOW COLUMNS 查看表结构的变化：

```
ALTER TABLE testalter_tbl DROP i;
ALTER TABLE testalter_tbl ADD i INT FIRST;
ALTER TABLE testalter_tbl DROP i;
ALTER TABLE testalter_tbl ADD i INT AFTER c;
```

FIRST 和 AFTER 关键字只占用于 ADD 子句，所以如果你想重置数据表字段的位置就需要先使用 DROP 删除字段然后使用 ADD 来添加字段并设置位置。

修改字段类型及名称

如果需要修改字段类型及名称, 你可以在ALTER命令中使用 MODIFY 或 CHANGE 子句 。

例如，把字段 c 的类型从 CHAR(1) 改为 CHAR(10)，可以执行以下命令:

```
mysql> ALTER TABLE testalter_tbl MODIFY c CHAR(10);
```

使用 CHANGE 子句, 语法有很大的不同。 在 CHANGE 关键字之后， 紧跟着的是你要修改的字段名，然后指定新字段的类型及名称。尝试如下实例：

```
mysql> ALTER TABLE testalter_tbl CHANGE i j BIGINT;
```

```
mysql> ALTER TABLE testalter_tbl CHANGE j j INT;
```

ALTER TABLE 对 Null 值和默认值的影响

当你修改字段时，你可以指定是否包含只或者是否设置默认值。

以下实例，指定字段 j 为 NOT NULL 且默认值为100 。

```
mysql> ALTER TABLE testalter_tbl
    -> MODIFY j BIGINT NOT NULL DEFAULT 100;
```

如果你不设置默认值，MySQL会自动设置该字段默认为 NULL。

修改字段默认值

你可以使用 ALTER 来修改字段的默认值，尝试以下实例：

```
mysql> ALTER TABLE testalter_tbl ALTER i SET DEFAULT 1000;
mysql> SHOW COLUMNS FROM testalter_tbl;
+-------+---------+------+-----+---------+-------+
| Field | Type    | Null | Key | Default | Extra |
+-------+---------+------+-----+---------+-------+
| c     | char(1) | YES  |     | NULL    |       |
| i     | int(11) | YES  |     | 1000    |       |
+-------+---------+------+-----+---------+-------+
2 rows in set (0.00 sec)
```

你也可以使用 ALTER 命令及 DROP子句来删除字段的默认值，如下实例：

```
mysql> ALTER TABLE testalter_tbl ALTER i DROP DEFAULT;
mysql> SHOW COLUMNS FROM testalter_tbl;
+-------+---------+------+-----+---------+-------+
| Field | Type    | Null | Key | Default | Extra |
+-------+---------+------+-----+---------+-------+
| c     | char(1) | YES  |     | NULL    |       |
| i     | int(11) | YES  |     | NULL    |       |
+-------+---------+------+-----+---------+-------+
2 rows in set (0.00 sec)
Changing a Table Type:
```

修改数据表类型，可以使用 ALTER 命令及 TYPE 子句来完成。尝试以下实例，我们将表 testalter_tbl 的类型修改为 MYISAM ：

注意：查看数据表类型可以使用 SHOW TABLE STATUS 语句。

```
mysql> ALTER TABLE testalter_tbl TYPE = MYISAM;
mysql>  SHOW TABLE STATUS LIKE 'testalter_tbl'\G
*************************** 1\. row ****************
           Name: testalter_tbl
           Type: MyISAM
     Row_format: Fixed
           Rows: 0
 Avg_row_length: 0
    Data_length: 0
Max_data_length: 25769803775
   Index_length: 1024
      Data_free: 0
 Auto_increment: NULL
    Create_time: 2007-06-03 08:04:36
    Update_time: 2007-06-03 08:04:36
     Check_time: NULL
 Create_options:
        Comment:
1 row in set (0.00 sec)
```

修改表名

如果需要修改数据表的名称，可以在 ALTER TABLE 语句中使用 RENAME 子句来实现。

尝试以下实例将数据表 testalter_tbl 重命名为 alter_tbl：

```
mysql> ALTER TABLE testalter_tbl RENAME TO alter_tbl;
```

ALTER 命令还可以用来创建及删除MySQL数据表的索引，该功能我们会在接下来的章节中介绍。

MySQL 索引

MySQL索引的建立对于MySQL的高效运行是很重要的，索引可以大大提高MySQL的检索速度。

打个比方，如果合理的设计且使用索引的MySQL是一辆兰博基尼的话，那么没有设计和使用索引的MySQL就是一个人力三轮车。

索引分单列索引和组合索引。单列索引，即一个索引只包含单个列，一个表可以有多个单列索引，但这不是组合索引。组合索引，即一个索包含多个列。

创建索引时，你需要确保该索引是应用在 SQL 查询语句的条件(一般作为 WHERE 子句的条件)。

实际上，索引也是一张表，该表保存了主键与索引字段，并指向实体表的记录。

上面都在说使用索引的好处，但过多的使用索引将会造成滥用。因此索引也会有它的缺点：虽然索引大大提高了查询速度，同时却会降低更新表的速度，如对表进行INSERT、UPDATE和DELETE。因为更新表时，MySQL不仅要保存数据，还要保存一下索引文件。

建立索引会占用磁盘空间的索引文件。

普通索引

创建索引

这是最基本的索引，它没有任何限制。它有以下几种创建方式：

```
CREATE INDEX indexName ON mytable(username(length));
```

如果是CHAR，VARCHAR类型，length可以小于字段实际长度；如果是BLOB和TEXT类型，必须指定 length。

修改表结构

```
ALTER mytable ADD INDEX [indexName] ON (username(length))
```

创建表的时候直接指定

```
CREATE TABLE mytable(

ID INT NOT NULL,

username VARCHAR(16) NOT NULL,

INDEX [indexName] (username(length))

);
```

删除索引的语法

```
DROP INDEX [indexName] ON mytable;
```

唯一索引

它与前面的普通索引类似，不同的就是：索引列的值必须唯一，但允许有空值。如果是组合索引，则列值的组合必须唯一。它有以下几种创建方式：

创建索引

```
CREATE UNIQUE INDEX indexName ON mytable(username(length))
```

修改表结构

```
ALTER mytable ADD UNIQUE [indexName] ON (username(length))
```

创建表的时候直接指定

```
CREATE TABLE mytable(

ID INT NOT NULL,

username VARCHAR(16) NOT NULL,

UNIQUE [indexName] (username(length))

);
```

使用**ALTER** 命令添加和删除索引

有四种方式来添加数据表的索引：

- **ALTER TABLE tbl_name ADD PRIMARY KEY (column_list):** 该语句添加一个主键，这意味着索引值必须是唯一的，且不能为NULL。
- **ALTER TABLE tbl_name ADD UNIQUE index_name (column_list):** 这条语句创建索引的值必须是唯一的（除了NULL外，NULL可能会出现多次）。
- **ALTER TABLE tbl_name ADD INDEX index_name (column_list):** 添加普通索引，索引值可出现多次。
- **ALTER TABLE tbl_name ADD FULLTEXT index_name (column_list):**该语句指定了索引为 FULLTEXT ，用于全文索引。

以下实例为在表中添加索引。

```
mysql> ALTER TABLE testalter_tbl ADD INDEX (c);
```

你还可以在 ALTER 命令中使用 DROP 子句来删除索引。尝试以下实例删除索引:

```
mysql> ALTER TABLE testalter_tbl DROP INDEX (c);
```

使用 **ALTER** 命令添加和删除主键

主键只能作用于一个列上，添加主键索引时，你需要确保该主键默认不为空（NOT NULL）。实例如下：

```
mysql> ALTER TABLE testalter_tbl MODIFY i INT NOT NULL;
mysql> ALTER TABLE testalter_tbl ADD PRIMARY KEY (i);
```

你也可以使用 ALTER 命令删除主键：

```
mysql> ALTER TABLE testalter_tbl DROP PRIMARY KEY;
```

删除指定时只需指定PRIMARY KEY，但在删除索引时，你必须知道索引名。

显示索引信息

你可以使用 SHOW INDEX 命令来列出表中的相关的索引信息。可以通过添加 \G 来格式化输出信息。

尝试以下实例:

```
mysql> SHOW INDEX FROM table_name\G
........
```

MySQL 临时表

MySQL 临时表在我们需要保存一些临时数据时是非常有用的。临时表只在当前连接可见，当关闭连接时，Mysql会自动删除表并释放所有空间。

临时表在MySQL 3.23版本中添加，如果你的MySQL版本低于 3.23版本就无法使用MySQL的临时表。不过现在一般很少有再使用这么低版本的MySQL数据库服务了。

MySQL临时表只在当前连接可见，如果你使用PHP脚本来创建MySQL临时表，那没当PHP脚本执行完成后，该临时表也会自动销毁。

如果你使用了其他MySQL客户端程序连接MySQL数据库服务器来创建临时表，那么只有在关闭客户端程序时才会销毁临时表，当然你也可以手动销毁。

实例

以下展示了使用MySQL 临时表的简单实例，以下的SQL代码可以适用于PHP脚本的mysql_query()函数。

```
mysql> CREATE TEMPORARY TABLE SalesSummary (
    -> product_name VARCHAR(50) NOT NULL
    -> , total_sales DECIMAL(12,2) NOT NULL DEFAULT 0.00
    -> , avg_unit_price DECIMAL(7,2) NOT NULL DEFAULT 0.00
    -> , total_units_sold INT UNSIGNED NOT NULL DEFAULT 0
);
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO SalesSummary
    -> (product_name, total_sales, avg_unit_price, total_units_sol
    -> VALUES
    -> ('cucumber', 100.25, 90, 2);

mysql> SELECT * FROM SalesSummary;
+--------------+-------------+----------------+------------------+
| product_name | total_sales | avg_unit_price | total_units_sold |
+--------------+-------------+----------------+------------------+
| cucumber     |      100.25 |          90.00 |                2 |
+--------------+-------------+----------------+------------------+
1 row in set (0.00 sec)
```

当你使用 **SHOW TABLES**命令显示数据表列表时，你将无法看到 SalesSummary 表。

如果你退出当前MySQL会话，再使用 **SELECT**命令来读取原先创建的临时表数据，那你会发现数据库中没有该表的存在，因为在你退出时该临时表已经被销毁了。

删除MySQL 临时表

默认情况下，当你断开与数据库的连接后，临时表就会自动被销毁。当然你也可以在当前MySQL会话使用 **DROP TABLE** 命令来手动删除临时表。

以下是手动删除临时表的实例：

```
mysql> CREATE TEMPORARY TABLE SalesSummary (
    -> product_name VARCHAR(50) NOT NULL
    -> , total_sales DECIMAL(12,2) NOT NULL DEFAULT 0.00
    -> , avg_unit_price DECIMAL(7,2) NOT NULL DEFAULT 0.00
    -> , total_units_sold INT UNSIGNED NOT NULL DEFAULT 0
);
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO SalesSummary
    -> (product_name, total_sales, avg_unit_price, total_units_sold
    -> VALUES
    -> ('cucumber', 100.25, 90, 2);

mysql> SELECT * FROM SalesSummary;
+--------------+-------------+----------------+------------------+
| product_name | total_sales | avg_unit_price | total_units_sold |
+--------------+-------------+----------------+------------------+
| cucumber     |      100.25 |          90.00 |                2 |
+--------------+-------------+----------------+------------------+
1 row in set (0.00 sec)
mysql> DROP TABLE SalesSummary;
mysql>  SELECT * FROM SalesSummary;
ERROR 1146: Table 'TUTORIALS.SalesSummary' doesn't exist
```

MySQL 复制表

如果我们需要完全的复制MySQL的数据表，包括表的结构，索引，默认值等。 如果仅仅使用**CREATE TABLE ... SELECT** 命令，是无法实现的。

本章节将为大家介绍如何完整的复制MySQL数据表，步骤如下：

- 使用 **SHOW CREATE TABLE** 命令获取创建数据表(**CREATE TABLE**) 语句，该语句包含了原数据表的结构，索引等。
- 复制以下命令显示的SQL语句，修改数据表名，并执行SQL语句，通过以上命令 将完全的复制数据表结构。
- 如果你想复制表的内容，你就可以使用 **INSERT INTO ... SELECT** 语句来实现。

实例

尝试以下实例来复制表 tutorials_tbl 。

步骤一：

获取数据表的完整结构。

```
mysql> SHOW CREATE TABLE tutorials_tbl \G;
*************************** 1\. row ***************************
       Table: tutorials_tbl
Create Table: CREATE TABLE `tutorials_tbl` (
  `tutorial_id` int(11) NOT NULL auto_increment,
  `tutorial_title` varchar(100) NOT NULL default '',
  `tutorial_author` varchar(40) NOT NULL default '',
  `submission_date` date default NULL,
  PRIMARY KEY  (`tutorial_id`),
  UNIQUE KEY `AUTHOR_INDEX` (`tutorial_author`)
) TYPE=MyISAM
1 row in set (0.00 sec)

ERROR:
No query specified
```

步骤二：

修改SQL语句的数据表名，并执行SQL语句。

```
mysql> CREATE TABLE `clone_tbl` (
  -> `tutorial_id` int(11) NOT NULL auto_increment,
  -> `tutorial_title` varchar(100) NOT NULL default '',
  -> `tutorial_author` varchar(40) NOT NULL default '',
  -> `submission_date` date default NULL,
  -> PRIMARY KEY  (`tutorial_id`),
  -> UNIQUE KEY `AUTHOR_INDEX` (`tutorial_author`)
-> ) TYPE=MyISAM;
Query OK, 0 rows affected (1.80 sec)
```

步骤三：

执行完第二步骤后，你将在数据库中创建新的克隆表 clone_tbl。 如果你想拷贝数据表的数据你可以使用 **INSERT INTO... SELECT** 语句来实现。

```
mysql> INSERT INTO clone_tbl (tutorial_id,
    ->                        tutorial_title,
    ->                        tutorial_author,
    ->                        submission_date)
    -> SELECT tutorial_id,tutorial_title,
    ->        tutorial_author,submission_date,
    -> FROM tutorials_tbl;
Query OK, 3 rows affected (0.07 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

执行以上步骤后，你将完整的复制表，包括表结构及表数据。

'Attr.nodeValue' is deprecated. Please use 'value' instead. adsbygoogle.js:32

MySQL 元数据

你可能想知道MySQL以下三种信息：

- 查询结果信息： SELECT, UPDATE 或 DELETE语句影响的记录数。
- 数据库和数据表的信息： 包含了数据库及数据表的结构信息。
- **MySQL**服务器信息： 包含了数据库服务器的当前状态，版本号等。

在MySQL的命令提示符中，我们可以很容易的获取以上服务器信息。 但如果使用Perl或PHP等脚本语言，你就需要调用特定的接口函数来获取。 接下来我们会详细介绍。

获取查询语句影响的记录数

PERL 实例

在 DBI 脚本中， 语句影响的记录数通过函数 do( ) 或 execute( )返回：

```
# 方法 1
# 使用do( ) 执行  $query
my $count = $dbh->do ($query);
# 如果发生错误会输出 0
printf "%d rows were affected\n", (defined ($count) ? $count : 0);

# 方法 2
# 使用prepare( ) 及 execute( ) 执行  $query
my $sth = $dbh->prepare ($query);
my $count = $sth->execute ( );
printf "%d rows were affected\n", (defined ($count) ? $count : 0);
```

PHP 实例

在PHP中，你可以使用 mysql_affected_rows( ) 函数来获取查询语句影响的记录数。

```
$result_id = mysql_query ($query, $conn_id);
# 如果查询失败返回
$count = ($result_id ? mysql_affected_rows ($conn_id) : 0);
print ("$count rows were affected\n");
```

数据库和数据表列表

你可以很容易的在MySQL服务器中获取数据库和数据表列表。 如果你没有足够的权限，结果将返回 null。

你也可以使用 SHOW TABLES 或 SHOW DATABASES 语句来获取数据库和数据表列表。

PERL 实例

```
# 获取当前数据库中所有可用的表。
my @tables = $dbh->tables ( );
foreach $table (@tables ){
   print "Table Name $table\n";
}
```

PHP 实例

```
<?php
$con = mysql_connect("localhost", "userid", "password");
if (!$con)
{
  die('Could not connect: ' . mysql_error());
}

$db_list = mysql_list_dbs($con);

while ($db = mysql_fetch_object($db_list))
{
  echo $db->Database . "<br />";
}
mysql_close($con);
?>
```

获取服务器元数据

以下命令语句可以在MySQL的命令提示符使用，也可以在脚本中 使用，如PHP脚本。

命令	描述
SELECT VERSION( )	服务器版本信息
SELECT DATABASE( )	当前数据库名 (或者返回空)
SELECT USER( )	当前用户名
SHOW STATUS	服务器状态
SHOW VARIABLES	服务器配置变量

MySQL 序列使用

MySQL序列是一组整数：1, 2, 3, ...，由于一张数据表只能有一个字段自增主键，如果你想实现其他字段也实现自动增加，就可以使用MySQL序列来实现。

本章我们将介绍如何使用MySQL的序列。

使用AUTO_INCREMENT

MySQL中最简单使用序列的方法就是使用 MySQL AUTO_INCREMENT 来定义列。

实例

以下实例中创建了数据表insect， insect中id无需指定值可实现自动增长。

```
mysql> CREATE TABLE insect
    -> (
    -> id INT UNSIGNED NOT NULL AUTO_INCREMENT,
    -> PRIMARY KEY (id),
    -> name VARCHAR(30) NOT NULL, # type of insect
    -> date DATE NOT NULL, # date collected
    -> origin VARCHAR(30) NOT NULL # where collected
);
Query OK, 0 rows affected (0.02 sec)
mysql> INSERT INTO insect (id,name,date,origin) VALUES
    -> (NULL,'housefly','2001-09-10','kitchen'),
    -> (NULL,'millipede','2001-09-10','driveway'),
    -> (NULL,'grasshopper','2001-09-10','front yard');
Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0
mysql> SELECT * FROM insect ORDER BY id;
+----+-------------+------------+------------+
| id | name        | date       | origin     |
+----+-------------+------------+------------+
|  1 | housefly    | 2001-09-10 | kitchen    |
|  2 | millipede   | 2001-09-10 | driveway   |
|  3 | grasshopper | 2001-09-10 | front yard |
+----+-------------+------------+------------+
3 rows in set (0.00 sec)
```

获取AUTO_INCREMENT值

在MySQL的客户端中你可以使用 SQL中的LAST_INSERT_ID( ) 函数来获取最后的插入表中的自增列的值。

在PHP或PERL脚本中也提供了相应的函数来获取最后的插入表中的自增列的值。

PERL实例

使用 mysql_insertid 属性来获取 AUTO_INCREMENT 的值。 实例如下：

```
$dbh->do ("INSERT INTO insect (name,date,origin)
VALUES('moth','2001-09-14','windowsill')");
my $seq = $dbh->{mysql_insertid};
```

PHP实例

PHP 通过 mysql_insert_id ()函数来获取执行的插入SQL语句中AUTO_INCREMENT列的值。

```
mysql_query ("INSERT INTO insect (name,date,origin)
VALUES('moth','2001-09-14','windowsill')", $conn_id);
$seq = mysql_insert_id ($conn_id);
```

重置序列

如果你删除了数据表中的多条记录，并希望对剩下数据的AUTO_INCREMENT列进行重新排列，那么你可以通过删除自增的列，然后重新添加来实现。 不过该操作要非常小心，如果在删除的同时又有新记录添加，有可能会出现数据混乱。操作如下所示：

```
mysql> ALTER TABLE insect DROP id;
mysql> ALTER TABLE insect
    -> ADD id INT UNSIGNED NOT NULL AUTO_INCREMENT FIRST,
    -> ADD PRIMARY KEY (id);
```

设置序列的开始值

一般情况下序列的开始值为1，但如果你需要指定一个开始值100，那我们可以通过以下语句来实现：

```
mysql> CREATE TABLE insect
    -> (
    -> id INT UNSIGNED NOT NULL AUTO_INCREMENT = 100,
    -> PRIMARY KEY (id),
    -> name VARCHAR(30) NOT NULL, # type of insect
    -> date DATE NOT NULL, # date collected
    -> origin VARCHAR(30) NOT NULL # where collected
);
```

或者你也可以在表创建成功后，通过以下语句来实现：

```
mysql> ALTER TABLE t AUTO_INCREMENT = 100;
```

MySQL 处理重复数据

有些 MySQL 数据表中可能存在重复的记录，有些情况我们允许重复数据的存在，但有时候我们也需要删除这些重复的数据。

本章节我们将为大家介绍如何防止数据表出现重复数据及如何删除数据表中的重复数据。

防止表中出现重复数据

你可以在MySQL数据表中设置指定的字段为 **PRIMARY KEY**（主键） 或者 **UNIQUE**（唯一） 索引来保证数据的唯一性。

让我们尝试一个实例：下表中无索引及主键，所以该表允许出现多条重复记录。

```
CREATE TABLE person_tbl
(
    first_name CHAR(20),
    last_name CHAR(20),
    sex CHAR(10)
);
```

如果你想设置表中字段first_name，last_name数据不能重复，你可以设置双主键模式来设置数据的唯一性， 如果你设置了双主键，那么那个键的默认值不能为NULL，可设置为NOT NULL。如下所示：

```
CREATE TABLE person_tbl
(
   first_name CHAR(20) NOT NULL,
   last_name CHAR(20) NOT NULL,
   sex CHAR(10),
   PRIMARY KEY (last_name, first_name)
);
```

如果我们设置了唯一索引，那么在插入重复数据时，SQL语句将无法执行成功,并抛出错。

INSERT IGNORE INTO与INSERT INTO的区别就是INSERT IGNORE会忽略数据库中已经存在的数据，如果数据库没有数据，就插入新的数据，如果有数据的话就跳过这条数据。这样就可以保留数据库中已经存在数据，达到在间隙中插入数据的目的。

以下实例使用了INSERT IGNORE INTO，执行后不会出错，也不会向数据表中插入重复数据：

```
mysql> INSERT IGNORE INTO person_tbl (last_name, first_name)
    -> VALUES( 'Jay', 'Thomas');
Query OK, 1 row affected (0.00 sec)
mysql> INSERT IGNORE INTO person_tbl (last_name, first_name)
    -> VALUES( 'Jay', 'Thomas');
Query OK, 0 rows affected (0.00 sec)
```

INSERT IGNORE INTO当插入数据时，在设置了记录的唯一性后，如果插入重复数据，将不返回错误，只以警告形式返回。 而REPLACE INTO into如果存在primary 或 unique相同的记录，则先删除掉。再插入新记录。

另一种设置数据的唯一性方法是添加一个UNIQUE索引，如下所示：

```
CREATE TABLE person_tbl
(
   first_name CHAR(20) NOT NULL,
   last_name CHAR(20) NOT NULL,
   sex CHAR(10)
   UNIQUE (last_name, first_name)
);
```

统计重复数据

以下我们将统计表中 first_name 和 last_name的重复记录数：

```
mysql> SELECT COUNT(*) as repetitions, last_name, first_name
    -> FROM person_tbl
    -> GROUP BY last_name, first_name
    -> HAVING repetitions > 1;
```

以上查询语句将返回 person_tbl 表中重复的记录数。 一般情况下，查询重复的值，请执行以下操作：

- 确定哪一列包含的值可能会重复。
- 在列选择列表使用COUNT(*)列出的那些列。
- 在GROUP BY子句中列出的列。
- HAVING子句设置重复数大于1。

过滤重复数据

如果你需要读取不重复的数据可以在 SELECT 语句中使用 DISTINCT 关键字来过滤重复数据。

```
mysql> SELECT DISTINCT last_name, first_name
    -> FROM person_tbl
    -> ORDER BY last_name;
```

你也可以使用 GROUP BY 来读取数据表中不重复的数据：

```
mysql> SELECT last_name, first_name
    -> FROM person_tbl
    -> GROUP BY (last_name, first_name);
```

删除重复数据

如果你想删除数据表中的重复数据，你可以使用以下的SQL语句：

```
mysql> CREATE TABLE tmp SELECT last_name, first_name, sex
    ->                  FROM person_tbl;
    ->                  GROUP BY (last_name, first_name);
mysql> DROP TABLE person_tbl;
mysql> ALTER TABLE tmp RENAME TO person_tbl;
```

当然你也可以在数据表中添加 INDEX（索引） 和 PRIMAY KEY（主键）这种简单的方法来删除表中的重复记录。方法如下：

```
mysql> ALTER IGNORE TABLE person_tbl
    -> ADD PRIMARY KEY (last_name, first_name);
```

MySQL 及 SQL 注入

如果您通过网页获取用户输入的数据并将其插入一个MySQL数据库，那么就有可能发生SQL注入安全的问题。

本章节将为大家介绍如何防止SQL注入，并通过脚本来过滤SQL中注入的字符。

所谓SQL注入，就是通过把SQL命令插入到Web表单递交或输入域名或页面请求的查询字符串，最终达到欺骗服务器执行恶意的SQL命令。

我们永远不要信任用户的输入，我们必须认定用户输入的数据都是不安全的，我们都需要对用户输入的数据进行过滤处理。

以下实例中，输入的用户名必须为字母、数字及下划线的组合，且用户名长度为 8 到 20 个字符之间：

```
if (preg_match("/^\w{8,20}$/", $_GET['username'], $matches))
{
   $result = mysql_query("SELECT * FROM users
                          WHERE username=$matches[0]");
}
 else
{
   echo "username 输入异常";
}
```

让我们看下在没有过滤特殊字符时，出现的SQL情况：

```
// 设定$name 中插入了我们不需要的SQL语句
$name = "Qadir'; DELETE FROM users;";
mysql_query("SELECT * FROM users WHERE name='{$name}'");
```

以上的注入语句中，我们没有对 $name 的变量进行过滤，$name 中插入了我们不需要的SQL语句，将删除 users 表中的所有数据。

在PHP中的 mysql_query() 是不允许执行多个SQL语句的，但是在 SQLite 和 PostgreSQL 是可以同时执行多条SQL语句的，所以我们对这些用户的数据需要进行严格的验证。

防止SQL注入，我们需要注意以下几个要点：

- 1.永远不要信任用户的输入。对用户的输入进行校验，可以通过正则表达式，或限制长度；对单引号和 双"-"进行转换等。
- 2.永远不要使用动态拼装sql，可以使用参数化的sql或者直接使用存储过程进行数据查询存取。
- 3.永远不要使用管理员权限的数据库连接，为每个应用使用单独的权限有限的

数据库连接。
- v 4.不要把机密信息直接存放，加密或者hash掉密码和敏感的信息。
- 5.应用的异常信息应该给出尽可能少的提示，最好使用自定义的错误信息对原始错误信息进行包装
- 6.sql注入的检测方法一般采取辅助软件或网站平台来检测，软件一般采用sql注入检测工具jsky，网站平台就有亿思网站安全平台检测工具。MDCSOFT SCAN等。采用MDCSOFT-IPS可以有效的防御SQL注入，XSS攻击等。

防止**SQL**注入

在脚本语言，如Perl和PHP你可以对用户输入的数据进行转义从而来防止SQL注入。

PHP的MySQL扩展提供了mysql_real_escape_string()函数来转义特殊的输入字符。

```
if (get_magic_quotes_gpc())
{
  $name = stripslashes($name);
}
$name = mysql_real_escape_string($name);
mysql_query("SELECT * FROM users WHERE name='{$name}'");
```

Like语句中的注入

like查询时，如果用户输入的值有"*"和"%"*，则会出现这种情况：用户本来只是想查询*"abcd*"，查询结果中却有"abcd_"、"abcde"、"abcdf"等等；用户要查询"30%"（注：百分之三十）时也会出现问题。

在PHP脚本中我们可以使用addcslashes()函数来处理以上情况，如下实例：

```
$sub = addcslashes(mysql_real_escape_string("%something_"), "%_");
// $sub == \%something\_
mysql_query("SELECT * FROM messages WHERE subject LIKE '{$sub}%'");
```

addcslashes() 函数在指定的字符前添加反斜杠。

语法格式:

```
addcslashes(string,characters)
```

参数	描述
string	必需。规定要检查的字符串。
characters	可选。规定受 addcslashes() 影响的字符或字符范围。

具体应用可以查看：PHP addcslashes() 函数

MySQL 导出数据

MySQL中你可以使用**SELECT...INTO OUTFILE**语句来简单的导出数据到文本文件上。

使用 SELECT ... INTO OUTFILE 语句导出数据

以下实例中我们将数据表 tutorials_tbl 数据导出到 /tmp/tutorials.txt 文件中:

```
mysql> SELECT * FROM tutorials_tbl
    -> INTO OUTFILE '/tmp/tutorials.txt';
```

你可以通过命令选项来设置数据输出的指定格式，以下实例为导出 CSV 格式：

```
mysql> SELECT * FROM passwd INTO OUTFILE '/tmp/tutorials.txt'
    -> FIELDS TERMINATED BY ',' ENCLOSED BY '"'
    -> LINES TERMINATED BY '\r\n';
```

在下面的例子中，生成一个文件，各值用逗号隔开。这种格式可以被许多程序使用。

```
SELECT a,b,a+b INTO OUTFILE '/tmp/result.text'
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
LINES TERMINATED BY '\n'
FROM test_table;
```

SELECT ... INTO OUTFILE 语句有以下属性:

- LOAD DATA INFILE是SELECT ... INTO OUTFILE的逆操作，SELECT句法。为了将一个数据库的数据写入一个文件，使用SELECT ... INTO OUTFILE，为了将文件读回数据库，使用LOAD DATA INFILE。
- SELECT...INTO OUTFILE 'file_name'形式的SELECT可以把被选择的行写入一个文件中。该文件被创建到服务器主机上，因此您必须拥有FILE权限，才能使用此语法。
- 输出不能是一个已存在的文件。防止文件数据被篡改。
- 你需要有一个登陆服务器的账号来检索文件。否则 SELECT ... INTO OUTFILE 不会起任何作用。
- 在UNIX中，该文件被创建后是可读的，权限由MySQL服务器所拥有。这意味着，虽然你就可以读取该文件，但可能无法将其删除。

导出表作为原始数据

mysqldump是mysql用于转存储数据库的实用程序。它主要产生一个SQL脚本，其中包含从头重新创建数据库所必需的命令CREATE TABLE INSERT等。

使用mysqldump导出数据需要使用 --tab 选项来指定导出文件指定的目录，该目标必须是可写的。

以下实例将数据表 tutorials_tbl 导出到 /tmp 目录中：

```
$ mysqldump -u root -p --no-create-info \
            --tab=/tmp TUTORIALS tutorials_tbl
password ******
```

导出SQL格式的数据

导出SQL格式的数据到指定文件，如下所示：

```
$ mysqldump -u root -p TUTORIALS tutorials_tbl > dump.txt
password ******
```

以上命令创建的文件内容如下：

```
-- MySQL dump 8.23
--
-- Host: localhost    Database: TUTORIALS
---------------------------------------------------------
-- Server version       3.23.58

--
-- Table structure for table `tutorials_tbl`
--

CREATE TABLE tutorials_tbl (
  tutorial_id int(11) NOT NULL auto_increment,
  tutorial_title varchar(100) NOT NULL default '',
  tutorial_author varchar(40) NOT NULL default '',
  submission_date date default NULL,
  PRIMARY KEY  (tutorial_id),
  UNIQUE KEY AUTHOR_INDEX (tutorial_author)
) TYPE=MyISAM;

--
-- Dumping data for table `tutorials_tbl`
--

INSERT INTO tutorials_tbl
       VALUES (1,'Learn PHP','John Poul','2007-05-24');
INSERT INTO tutorials_tbl
       VALUES (2,'Learn MySQL','Abdul S','2007-05-24');
INSERT INTO tutorials_tbl
       VALUES (3,'JAVA Tutorial','Sanjay','2007-05-06');
```

如果你需要导出整个数据库的数据，可以使用以下命令：

```
$ mysqldump -u root -p TUTORIALS > database_dump.txt
password ******
```

如果需要备份所有数据库，可以使用以下命令：

```
$ mysqldump -u root -p --all-databases > database_dump.txt
password ******
```

--all-databases 选项在 MySQL 3.23.12 及以后版本加入。

该方法可用于实现数据库的备份策略。

将数据表及数据库拷贝至其他主机

如果你需要将数据拷贝至其他的 MySQL 服务器上, 你可以在 mysqldump 命令中指定数据库名及数据表。

在源主机上执行以下命令，将数据备份至 dump.txt 文件中:

```
$ mysqldump -u root -p database_name table_name > dump.txt
password *****
```

如果完整备份数据库，则无需使用特定的表名称。

如果你需要将备份的数据库导入到MySQL服务器中，可以使用以下命令，使用以下命令你需要确认数据库已经创建：

```
$ mysql -u root -p database_name < dump.txt
password *****
```

```
你也可以使用以下命令将导出的数据直接导入到远程的服务器上，但请确保两台服务器是相
$ mysqldump -u root -p database_name \
       | mysql -h other-host.com database_name
```

以上命令中使用了管道来将导出的数据导入到指定的远程主机上。

MySQL 导入数据

MySQL中可以使用两种简单的方式来导入MySQL导出的数据。

使用 LOAD DATA 导入数据

MySQL 中提供了LOAD DATA INFILE语句来插入数据。 以下实例中将从当前目录中读取文件 dump.txt ，将该文件中的数据插入到当前数据库的 mytbl 表中。

```
mysql> LOAD DATA LOCAL INFILE 'dump.txt' INTO TABLE mytbl;
```

如果指定LOCAL关键词，则表明从客户主机上按路径读取文件。如果没有指定，则文件在服务器上按路径读取文件。

你能明确地在LOAD DATA语句中指出列值的分隔符和行尾标记，但是默认标记是定位符和换行符。

两个命令的 FIELDS 和 LINES 子句的语法是一样的。两个子句都是可选的，但是如果两个同时被指定，FIELDS 子句必须出现在 LINES 子句之前。

如果用户指定一个 FIELDS 子句，它的子句 （TERMINATED BY、[OPTIONALLY] ENCLOSED BY 和 ESCAPED BY) 也是可选的，不过，用户必须至少指定它们中的一个。

```
mysql> LOAD DATA LOCAL INFILE 'dump.txt' INTO TABLE mytbl
  -> FIELDS TERMINATED BY ':'
  -> LINES TERMINATED BY '\r\n';
```

LOAD DATA 默认情况下是按照数据文件中列的顺序插入数据的，如果数据文件中的列与插入表中的列不一致，则需要指定列的顺序。

如，在数据文件中的列顺序是 a,b,c，但在插入表的列顺序为b,c,a，则数据导入语法如下：

```
mysql> LOAD DATA LOCAL INFILE 'dump.txt'
    -> INTO TABLE mytbl (b, c, a);
```

使用 mysqlimport 导入数据

mysqlimport客户端提供了LOAD DATA INFILEQL语句的一个命令行接口。mysqlimport的大多数选项直接对应LOAD DATA INFILE子句。

从文件 dump.txt 中将数据导入到 mytbl 数据表中, 可以使用以下命令：

```
$ mysqlimport -u root -p --local database_name dump.txt
password *****
```

mysqlimport命令可以指定选项来设置指定格式,命令语句格式如下：

```
$ mysqlimport -u root -p --local --fields-terminated-by=":" \
   --lines-terminated-by="\r\n"  database_name dump.txt
password *****
```

mysqlimport 语句中使用 --columns 选项来设置列的顺序：

```
$ mysqlimport -u root -p --local --columns=b,c,a \
    database_name dump.txt
password *****
```

mysqlimport的常用选项介绍

选项	功能
-d or --delete	新数据导入数据表中之前删除数据数据表中的所有信息
-f or --force	不管是否遇到错误，mysqlimport将强制继续插入数据
-i or --ignore	mysqlimport跳过或者忽略那些有相同唯一 关键字的行， 导入文件中的数据将被忽略。
-l or -lock-tables	数据被插入之前锁住表，这样就防止了， 你在更新数据库时，用户的查询和更新受到影响。
-r or -replace	这个选项与－i选项的作用相反；此选项将替代 表中有相同唯一关键字的记录。
--fields-enclosed-by= char	指定文本文件中数据的记录时以什么括起的， 很多情况下 数据以双引号括起。 默认的情况下数据是没有被字符括起的。
--fields-terminated-by=char	指定各个数据的值之间的分隔符，在句号分隔的文件中， 分隔符是句号。您可以用此选项指定数据之间的分隔符。 默认的分隔符是跳格符（Tab）
--lines-terminated-by=str	此选项指定文本文件中行与行之间数据的分隔字符串 或者字符。 默认的情况下mysqlimport以newline为行分隔符。 您可以选择用一个字符串来替代一个单个的字符： 一个新行或者一个回车。

mysqlimport命令常用的选项还有-v 显示版本（version）， -p 提示输入密码（password）等。

免责声明

W3School提供的内容仅用于培训。我们不保证内容的正确性。通过使用本站内容随之而来的风险与本站无关。W3School 简体中文版的所有内容仅供测试，对任何法律问题及风险不承担任何责任。